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 problem 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-consistency*" 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 compound 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 D_x
            IF NOT satisfies((<x,v>), C_x)
            THEN D_x ← D_x – {v};
    return(Z, D, C);                    /* certain D_x 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 | C_x,y ∈ C}
            /* x→y is an arc; C_y,x is the same object as C_x,y */
    REPEAT
        Changed ← False;
        FOR each x→y ∈ Q DO
            Changed ← (Revise_Domain(x→y, (Z, D, C)) OR
                Changed);
            /* side effect of Revise_Domain: D_x 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 $x \rightarrow y$ and $y \rightarrow x$ are put into Q. AC-1 examines every $x \rightarrow y$ 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($x \rightarrow y$, (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($x \rightarrow y$, (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 | C_{x,y} ∈ C};
          /* x→y is an arc; C_{y,x} is the same object as C_{x,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 | C_{z,x} ∈ C ∧ z ≠ x ∧ z ≠ y};
                /* side effect of Revise_Domain: D_x 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 $z \rightarrow x$ (except when $z = y$) is added to the queue Q if *Revise_Domain*$(x \rightarrow y, (Z, D, C))$ returns *True*. $y \rightarrow x$ 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 $\Omega(a^2e)$.

In the worst case, each call of Revise_Domain deletes one value from a domain. Each arc $x \rightarrow y$ 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.

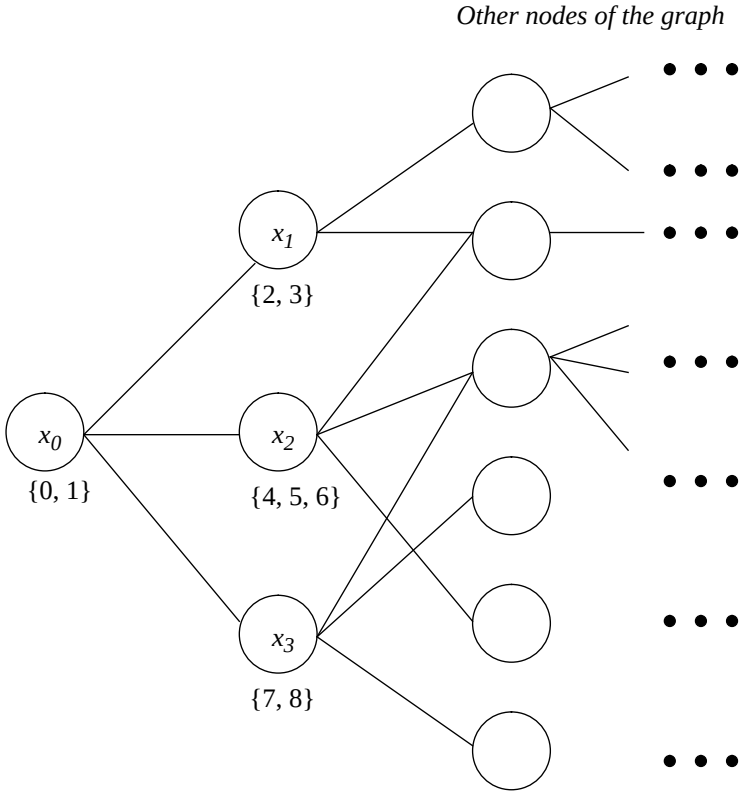


Figure 4.1 Example of a partial constraint graph. Constraints: sum of the values for the constrained variables must be even

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

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

This counter stores the support to the label $<x_0,0>$ 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 x_j where j < i AND C_i,j ∈ C DO
            Revise_Domain(x_j→x_i, (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 $O(a^2e)$, 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 $O(a^2n)$.

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*".

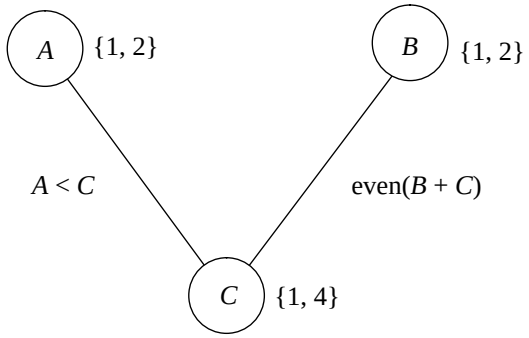


Figure 4.2 An example showing that running DAC on both directions for an arbitrary ordering does not achieve AC (After achieving DAC for both orderings (*A*, *B*, *C*) and (*C*, *B*, *A*), only <*C*,1> will be removed, but *C* has no compatible values with <*B*,1>, which means <*B*,1> should have been 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 $A \neq B$, 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:

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

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|; Y^n ← C;
    REPEAT
        Y^0 ← Y^n;
        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 Y^n = Y^0;
    C ← Y^n;
    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: C_{i,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 C_{i,j} ∈ C
      FOR k = 1 TO n DO
         FOR each b ∈ D_i DO
            FOR each c ∈ D_j such that satisfies((<i,b><j,c>), C_{i,j})
               holds DO
               BEGIN
                  Total ← 0;
                  FOR each d ∈ D_k DO
                     IF (satisfies((<i,b><k,d>), C_{i,k}) & satis-
                        fies((<k,d><j,c>), C_{k,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 $O(n^3a^3)$, 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 $O(n^3a^2)$ counters, with each of which having a maximum of a values, the maximum number of times that the counters can be reduced is $O(n^3a^3)$. 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^2a^2 2-compound labels that one can delete, the WHILE loop in step 2 can only iterate $O(n^2a^2)$ 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 $O(n^3a^3)$. 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 $O(n^3a^3)$.

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

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 ⇔ z_i < z_j */
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 ((z_i→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 $O(n^3a^3)$.

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 $O(n^2a^2)$, 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 jth 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 jth 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 X^i = (x_1,...,x_i) of variables x_1 < ... < x_i DO
            FOR each i-tuple V^i = (v_1,v_2,...,v_i) of values v_1, ..., v_i DO
                BEGIN
                    FOR each y ∈ (Z − {x_1,x_2,...,x_i}) DO
                        Counter[X^i,V^i,y] ← |D_y|;
                    IF NOT satisfies((<x_1,v_1>...<x_i,v_i>), C_{x_1...x_i}) THEN
                        BEGIN Set ← Set + {(X^i,V^i,i)}; M[X^i , V^i] ← 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 (X^i,V^i,i) from Set, where X^i = (x_1,...,x_i) and V^i =
                (v_1,...,v_i);
            KS_Upward_Propagate(X^i, V^i, i, k);
            KS_Downward_Propagate(X^i, V^i, i, k)
        END
    return(Z, D, C);
END /* of KS-1 */


PROCEDURE KS_Upward_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_Upward_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' ∉ {x_1,x_2,...,x_i} DO
            BEGIN
                X^{i+1} ← (x_1,...,x_i,x'); V^{i+1} ← (v_1,...,v_i,v');
                IF (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_1...x_ix'} ← C_{x_1...x_ix'} −{(<x_1,v_1>...<x_i,v_i><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_Downward_Propagate examines i - 1 compound labels. Z, D, C, Counter, 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 = ($<x_1,v_1>$ ... $<x_i,v_i>$) 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(($<x_1,v_1>$...$<x_i,v_i><x_j,v_j>$), $C_{x_1 \ldots x_i x_j}$) 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:

$$\forall \text{ graph}((V, E)):$$
$$(V, E) = \text{G}((Z, D, C)) \equiv$$
$$((V = Z) \wedge E = \{(x,y) \mid x, y \in Z \wedge (\exists\, C_S \in C: x, y \in S)\}) \blacksquare$$

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:

$$\forall \text{ csp}((Z, D, C)):$$
$$(\forall <: \text{total\_ordering}(Z, <): (V, E) = \text{primal\_graph}((Z, D, C)):$$
$$(\forall x \in V: \text{parents}(x, (V, E), <) \equiv \{y \mid y < x \wedge (x,y) \in E\}\ )) \blacksquare$$

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:

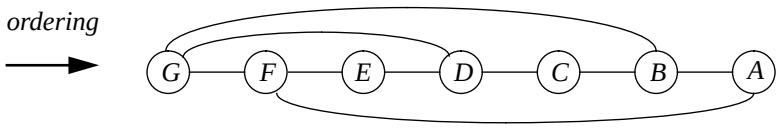
A CSP P is **adaptive-consistent** under a total-ordering of its variables if for all variables x, there exists a constraint C_S on the parents of x (S), and every compound label in C_S satisfies all the relevant constraints on S.

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

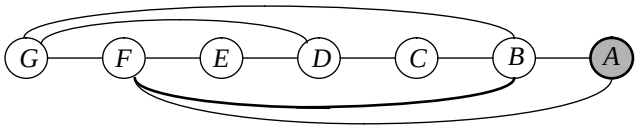
The concept of backtrack-free search involves an ordering of the variables (Definitions 1-28, 1-29). To achieve adaptive-consistency, the variables are processed according to the reverse of this ordering. For each variable x that is being processed, a k-constraint is created for its Parents, where k 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, <)
/* x_i 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 < x_i ∧ (∃ C_X ∈ C: w, x_i ∈ X)};
            C_S ← {cl | cl = compound label for S such that ∃ v_i ∈ D_{x_i}:
                satisfies(cl+<x_i,v_i>, CE(S + {x_i}, (Z, D, C)))};
            C ← C + {C_S};
        END;
    return(Z, D, C, <);
END /* of Adaptive_consistency */
```

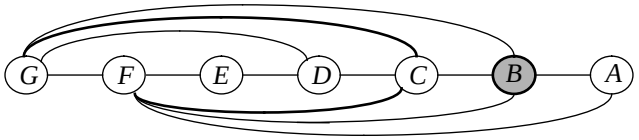
The Adaptive_consistency procedure assumes that the variables are given the ordering $x_1, x_2, ..., x_n$, where n is the number of variables in the problem. These variables are processed in reverse order. When x_i is processed, the procedure removes from the constraint for the parents of x_i all those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



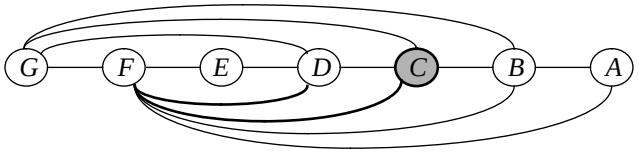
(a) Ordered graph to be processed (from Figure 3.5)
nodes order: (*G*,*F*,*E*,*D*,*C*,*B*,*A*), process order: *(A*,*B*,*C*,*D*,*E*,*F*,*G*)



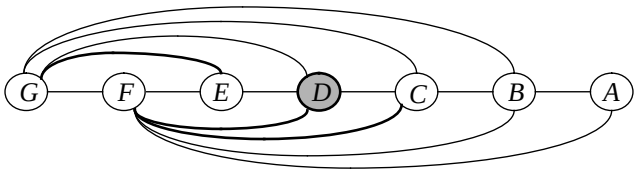
(b) Node *A* is processed, and edge (*F*, *B*) is added



(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 non-empty, 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:

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

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

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 <*x*,*a*>.

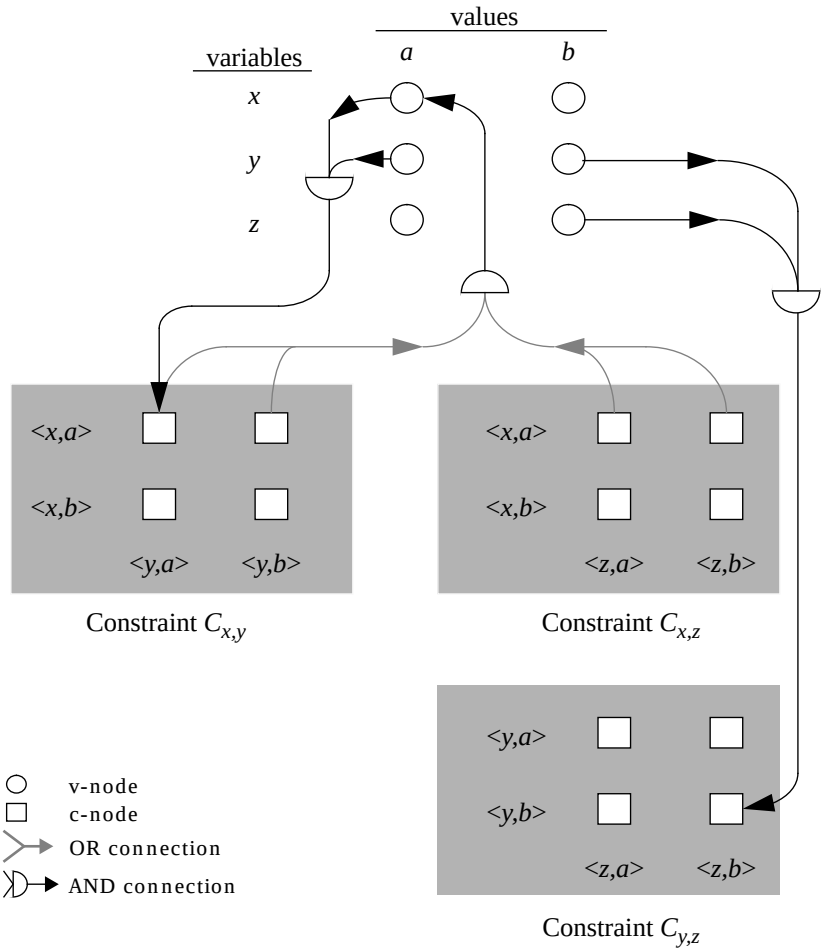


Figure 4.4 A connectionist representation of a binary CSP (*Z*, *D*, *C*), where the variables $Z = \{x, y, z\}$ and all the domains are $\{a, b\}$ (only three sets of connections are shown here)

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 $(\langle x_1,v_1\rangle\ \langle x_2,v_2\rangle)$ $s(\langle x_1,v_1\rangle\ \langle x_2,v_2\rangle)$.

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

$$\begin{aligned} v(\langle x,v\rangle) \leftarrow\ & \{s(\langle x,v\rangle\langle x',v'\rangle) \mid x' \in Z \,\&\, v' \in D_{x'} \,\&\, x \neq x'\} \cup \\ & \{s(\langle x_1,v_1\rangle\langle x_2,v_2\rangle) \mid \\ & \qquad x_1 \in Z \,\&\, v_1 \in D_{x_1} \,\&\, x_1 \neq x \,\&\, x_2 \in Z \,\&\, v_2 \in D_{x_2} \,\&\, x_2 \neq x\} \end{aligned}$$

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 $\langle x,a\rangle$ will be initialized to the set of signatures $\{s(\langle x,a\rangle\langle y,a\rangle), s(\langle x,a\rangle\langle y,b\rangle), s(\langle x,a\rangle\langle z,a\rangle), s(\langle x,a\rangle\langle z,b\rangle), s(\langle y,a\rangle\langle z,a\rangle), s(\langle y,a\rangle\langle z,b\rangle), s(\langle y,b\rangle\langle z,a\rangle), s(\langle y,b\rangle\langle z,b\rangle)\}$.

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: $\Omega(a^2e)$ 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.