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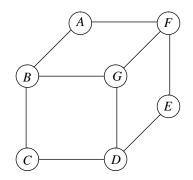
(1)   the *minimal width ordering* (MWO) heuristic — by exploiting the topology of the nodes in the primal graph of the problem (Definition 4-1), the MWO heuristic orders the variables before the search starts. The intention is to reduce the need for backtracking;

(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.

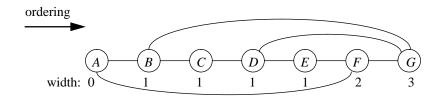### 6.2.1 The Minimal Width Ordering Heuristic

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.
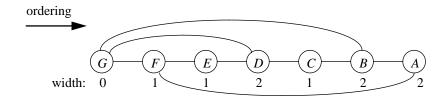
6.2.1.1 Definitions and motivation

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.
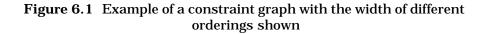
(a) A constraint graph to be labelled

ordering

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)

ordering

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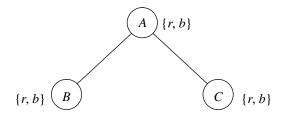
$$\forall \ csp((Z, D, C)): (\forall <: total\_ordering(Z, <):$$
$$strong \ k\text{-}consistent((Z, D, C)) \wedge width(G(Z, D, C), <) < k \Rightarrow$$
$$backtrack\text{-}free((Z, D, C), <)$$

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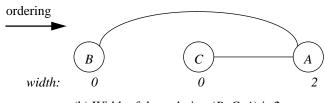
$$\forall \ csp((Z, D, C)):$$
$$strong \ k\text{-}consistent((Z, D, C)) \wedge width(G(Z, D, C)) < k \Rightarrow$$
$$(\exists <: total\_ordering(Z, <): backtrack\text{-}free((Z, D, C), <)$$
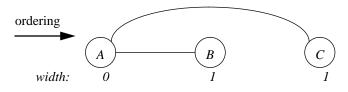
Proof

(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

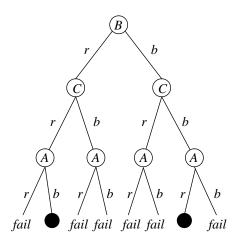(b) Width of the ordering (B, C, A) is 2

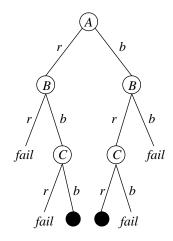(c) Width of the search ordering (A, B, C) is 1

(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
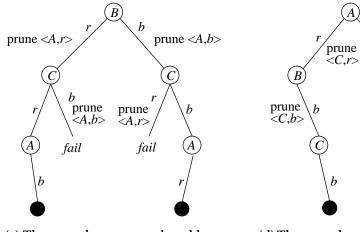
(c) The search space explored by FC under the ordering (*B, C, A*); number of branches explored: 8

(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.

$\forall$ graph(G): width(G) = 1 $\Leftrightarrow$ 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.
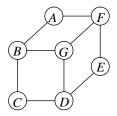
6.2.1.2 Finding minimal width ordering

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]:
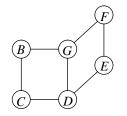
```
PROCEDURE Find_Minimal_Width_Ordering( (V, E) )
/* (V, E) is a graph where V, E are the set of nodes and edges,
    respectively */
BEGIN
    Q = ( );            /* Q is initialized to an empty sequence */
    REPEAT
        N ← the node in V joined by the least number of edges in E;
        /* in 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 */
```
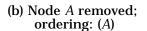
The Find_Minimal_Width_Ordering procedure returns a sequence which has the minimal width of the graph. In other words, the width of the returned ordering is the width of the constraint graph. The proof of this post-condition of the procedure will not be presented here; interested readers are referred 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, $\left| h(v) - h(w) \right|$ 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:

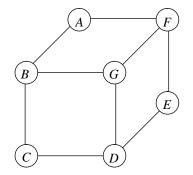The **bandwidth of a graph** is the minimal bandwidth of all orderings in the graph:

$\forall$ 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))

ordering

band-
width:

(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)

ordering

band-
width:

(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 chron-ological 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 algo-rithm. 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)) $\equiv$
      ($a \in Z \wedge b \in V \wedge b \notin Z$)))) ∎

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
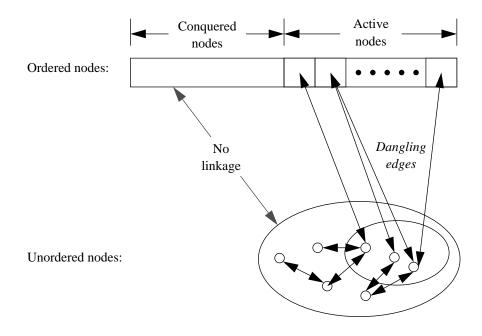
Figure 6.6 **Node partitioning in bandwidth determination**

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)
/* $r = (v_1, v_2, ..., v_i)$ */
/* 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 ((v_j 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((v_j, v_{j+1}, ..., v_i, 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* otherwise:

PROCEDURE **Plausible**((r,d), k)
/* $r = (v_1, v_2, ..., v_i)$, 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 (v_j 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.
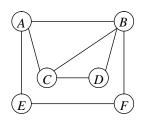
Determine_Bandwidth basically performs a breadth-first search in the space of active-region-dangling-edges pairs. Figure 6.7 shows part of the space searched by Determine_Bandwidth in an example graph.
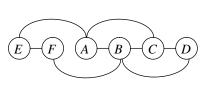
To find an ordering with the minimal bandwidth for a graph, one may call Determine_Bandwidth iteratively, increasing k by 1 at a time.

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:
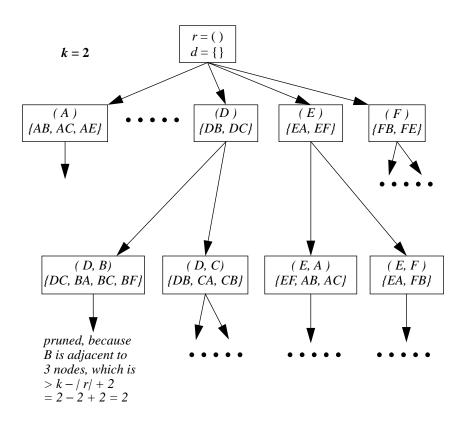
(1)   There are at most $O(n^k)$ pairs of (r, d) in which r consists of k nodes. With the help of T, each such pair is processed no more than once. When r has k nodes, *Update_active_area* and *Plausible* together ensure that there is exactly one edge connected to the first node of r. Therefore, the 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

(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*).
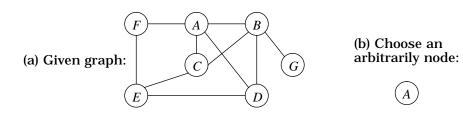
6.2.5 Finding the next variable to label

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.

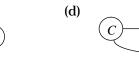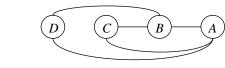In principle, the MWO heuristic may be useful for CSPs where:

(a) Given graph:

(b) Choose an arbitrarily node:
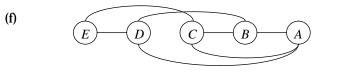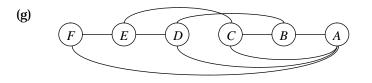
ordering

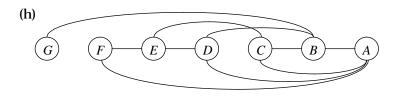(c)

(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;
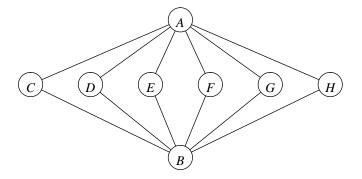
Point (2) suggests that the FFP is especially effective when used with lookahead algorithms.

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.
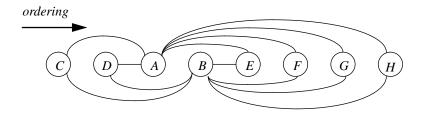
One could attempt to combine the MWO and MBO heuristics. Given a constraint graph, it is possible that more than one ordering has the minimum width and minimal bandwidth. For example, the bandwidth of both the ordered graph shown in Figure 6.5(b) and Figure 6.5(c) above have a width of 2, but the former 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.)
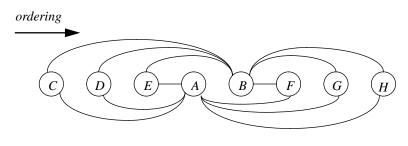
It is possible to combine the MWO and MBO heuristics with FFP. One way in

(a) The constraint graph to be ordered

ordering

(b) An ordering with minimal width, but not minimal
bandwidth (wdith = 2, bandwidth = 5)

ordering

(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 $\langle x,v \rangle$ 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 $\langle x,v' \rangle$ 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 variable, 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].