Chapter 5
Basic search strategies for solving CSPs

5.1 Introduction

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

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

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

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

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

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

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

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

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

5.2 General Search Strategies

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

5.2.1 Chronological backtracking

5.2.1.1 The BT algorithm

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

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

5.2.1.2 Implementation of BT

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

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

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

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

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

5.2.2 Iterative broadening

5.2.2.1 Observation and the IB algorithm

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

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

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

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


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

5.2.2.2 Discussions on IB

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

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

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

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

5.2.2.3 Implementation of IB

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

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

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

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

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

5.3 Lookahead Strategies

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

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

5.3.1 Forward Checking

5.3.1.1 The FC algorithm

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

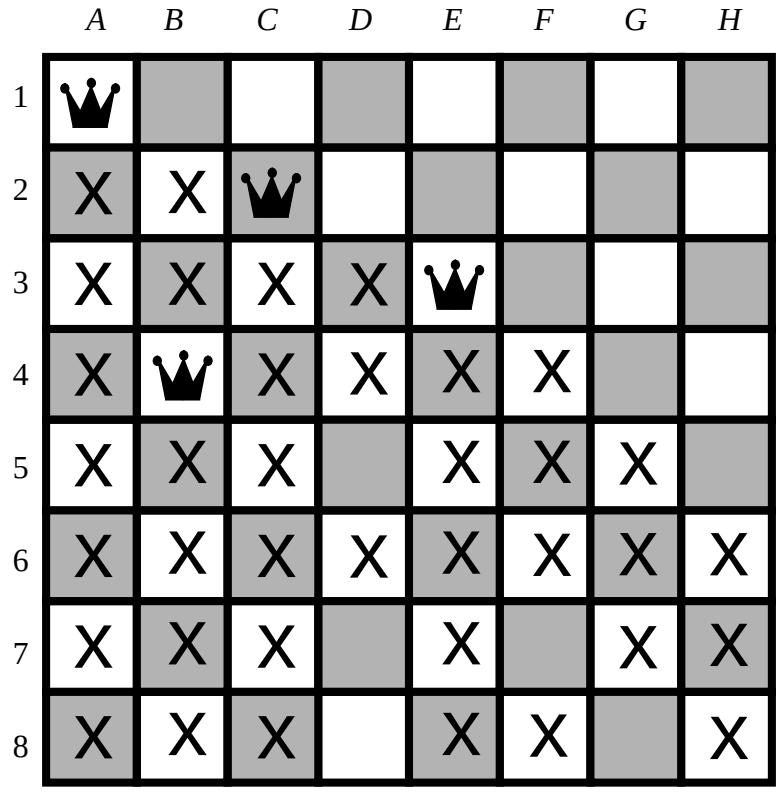


Figure 5.1 Example showing the effect of FC: the label <4,*B*> should be rejected because all the values for Queen 6 are incompatible with the committed labels

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

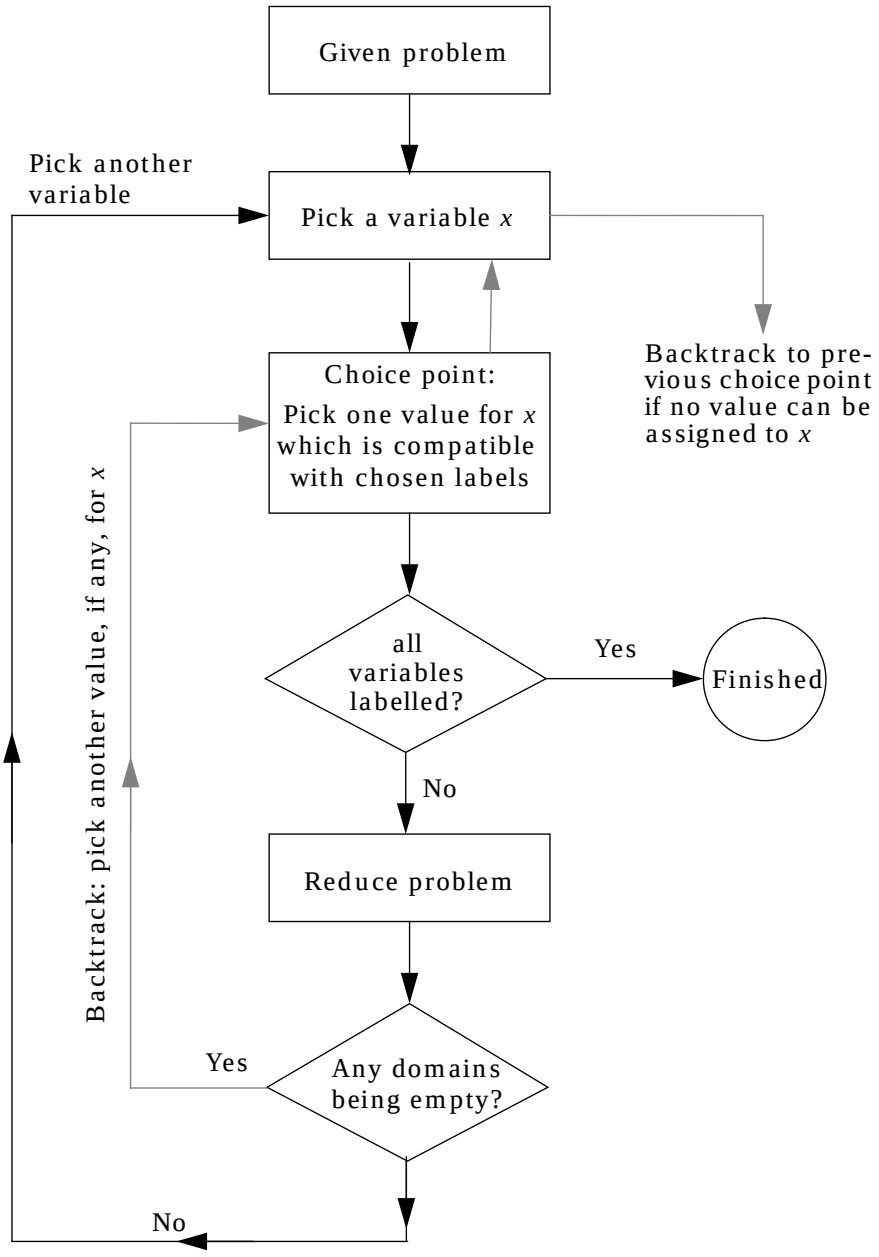


Figure 5.2 The control of lookahead algorithms

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


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


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

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

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

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

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


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

5.3.1.2 Implementation of FC

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

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

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

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

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

5.3.2 The Directional AC-Lookahead algorithm

5.3.2.1 The DAC-L algorithm

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

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


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

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

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

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

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

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

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

5.3.2.2 Implementation of DAC-L

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

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

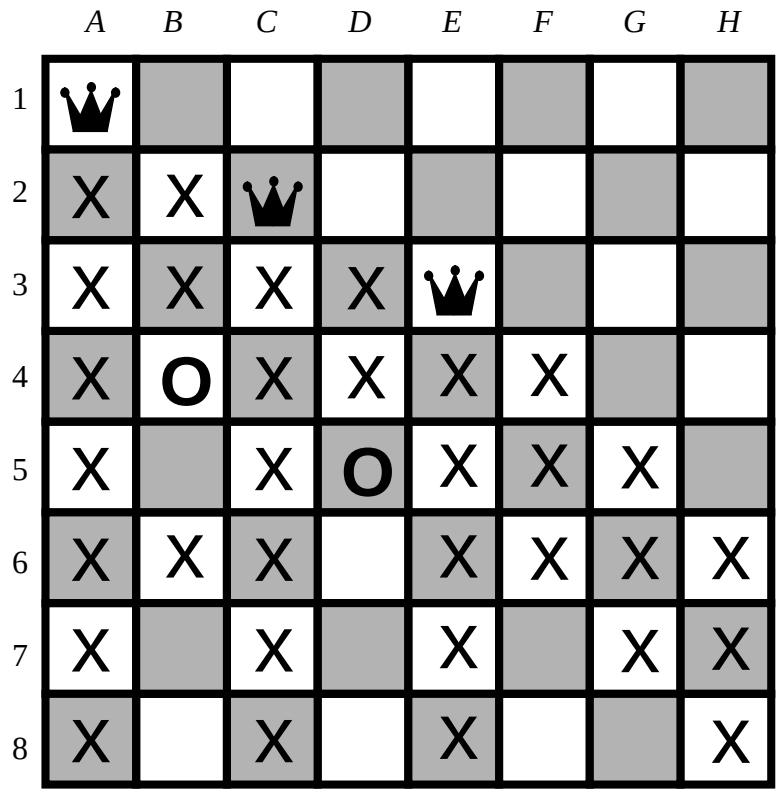


Figure 5.3 Example showing the behaviour of DAC-Lookahead: <4,*B*> will be rejected at this stage because all the available values for Queen 6 are incompatible with it (squares marked **o** are rejected after DAC is achieved)

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

5.3.3 The AC-Lookahead algorithm

5.3.3.1 The algorithm AC-L

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

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

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

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


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

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

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

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

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

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

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

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

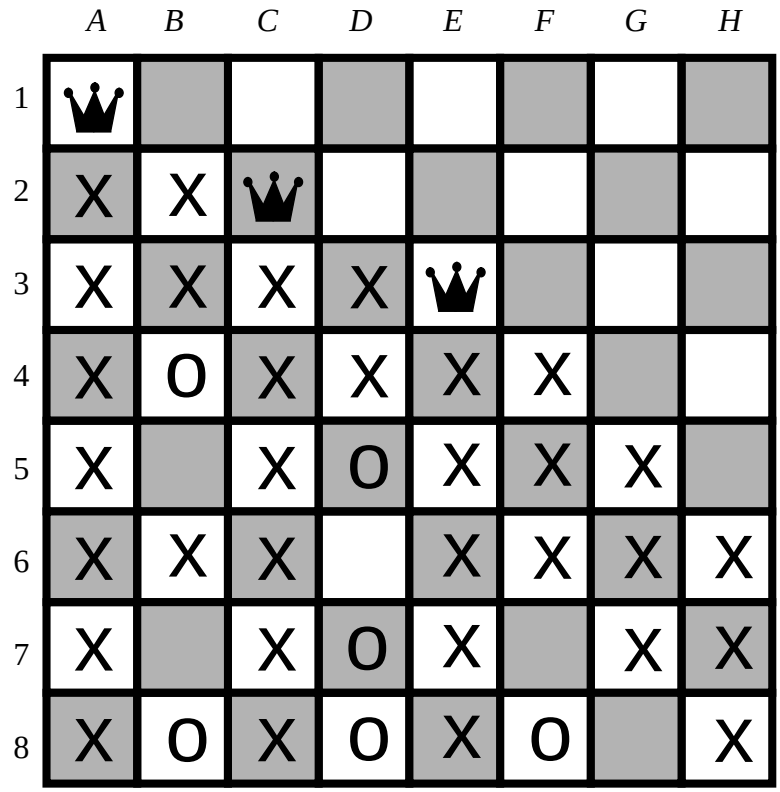


Figure 5.4 Example showing the behaviour of AC-Lookahead: label <3,*E*> will be rejected if AC is to be achieved. Labels marked **O** are rejected for having no compatible values in Queen 6; then label <5, *H*> is rejected for having no compatible value for Queen 4; then <7,*B*> will be removed for having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

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

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

5.3.3.2 Implementation of AC-Lookahead

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

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

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

5.3.4 Remarks on lookahead algorithms

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

5.4 Gather-information-while-searching Strategies

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

5.4.1 Dependency directed backtracking

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

5.4.1.1 BackJumping

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

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

The algorithm is formally described as follows:

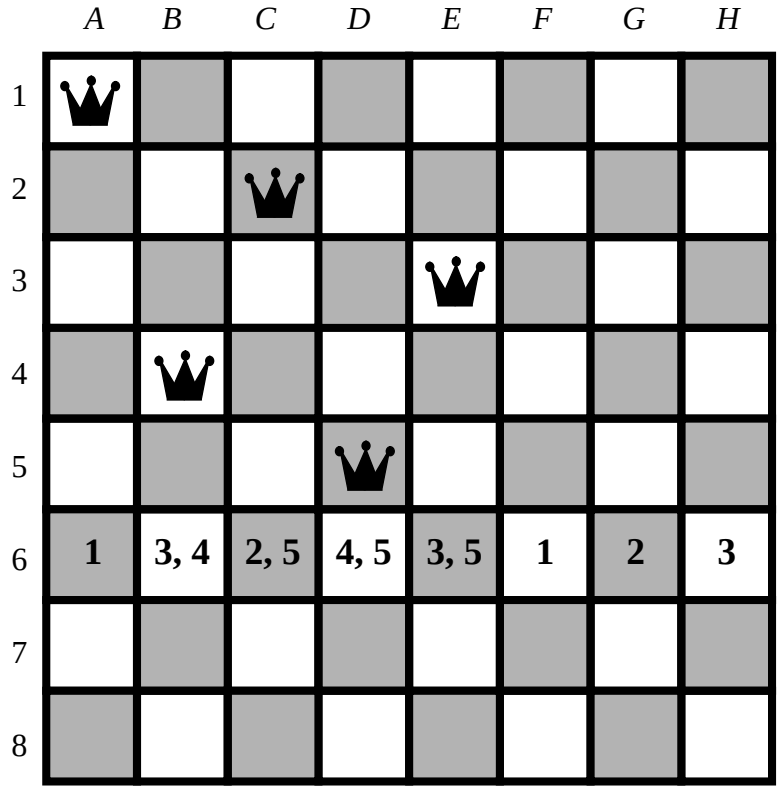


Figure 5.5 A board situation in the 8-queens problem. The numbers in row 6 indicate the labelled queens that the corresponding squares are incompatible with. It is possible at this stage to realize that changing the value of Queen 5 will not resolve the conflicts, or learn incompatible compound labels

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

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

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

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

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

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

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

.....

5.4.1.2 Implementation of BJ

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

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

5.4.1.3 Graph-based BackJumping

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

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

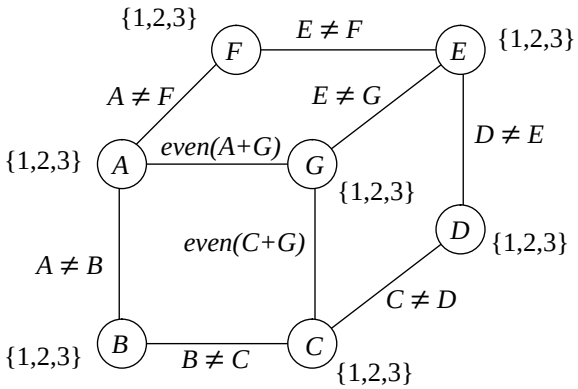
Figure 5.6 shows a hypothetical CSP. Figure 5.6(a) shows the constraint graph, where the nodes represent variable and the edges represent constraints. The variables in the problem are *A*, *B*, *C*, *D*, *E*, *F* and *G*. Assume that the domains for all the variables are {1, 2, 3}, and all the constraints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

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

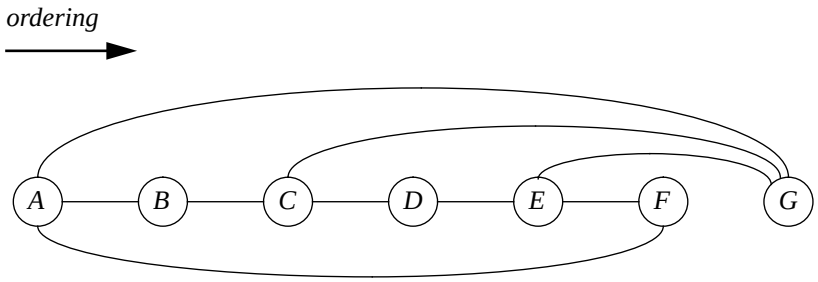
By making use of the constraints, graph-based backjumping manages to jump past the decisions which are definitely irrelevant to the failure. Using the constraints saves computational effort in the analysis. However, graph-based backjumping is not guaranteed to identify the real culprit at all times.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

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

Now no label for *G* satisfies all the constraints. As suggested, graph-based back-



(a) A constraint graph: variables are *A*, *B*, *C*, *D*, *E*, *F* and *G*; domains for all the variables are {1,2,3}; constraints are shown on the edges



(b) Ordering of the variables

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

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

5.4.2 Learning nogood compound labels algorithms

5.4.2.1 The LNCL algorithm

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

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

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

Definition 5-1:

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

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

Definition 5-2:

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

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

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


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


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

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


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

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

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

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

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

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

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

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

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

5.4.2.2 Implementation of LNCL

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

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

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

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

5.4.3 Backchecking and Backmarking

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

5.4.3.1 Backchecking

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

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

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

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

5.4.3.2 Backmarking

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

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

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

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

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

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

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

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

```
PROCEDURE Backmark-1( Z, D, C );
BEGIN
    i ← 1;
    For each x in Z DO
        BEGIN
            LowUnit(x) ← 1;
            FOR each v in Dx DO
                Mark(x,v) ← 1;
            Ordering(x) ← i; i ← i + 1;
        END /* end of initialization */
    BM-1( Z, { }, D, C, 1 );
END /* of Backmark-1 */
```

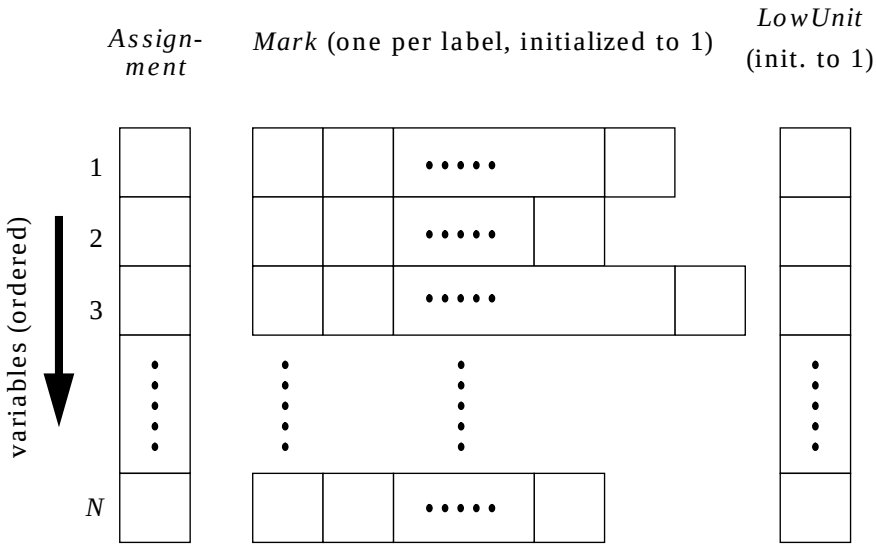


Figure 5.7 Variable sets used by Backmark-1. *Assignment*(x) = the value assigned to variable *x*. *Mark*(*x*,*v*) = the lowest level at which <*x*,*v*> failed. *LowUnit*(x) = the ordering of the lowest variable *y* which *Assignment* has been changed since the last time *x* is visited

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

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


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

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

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

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

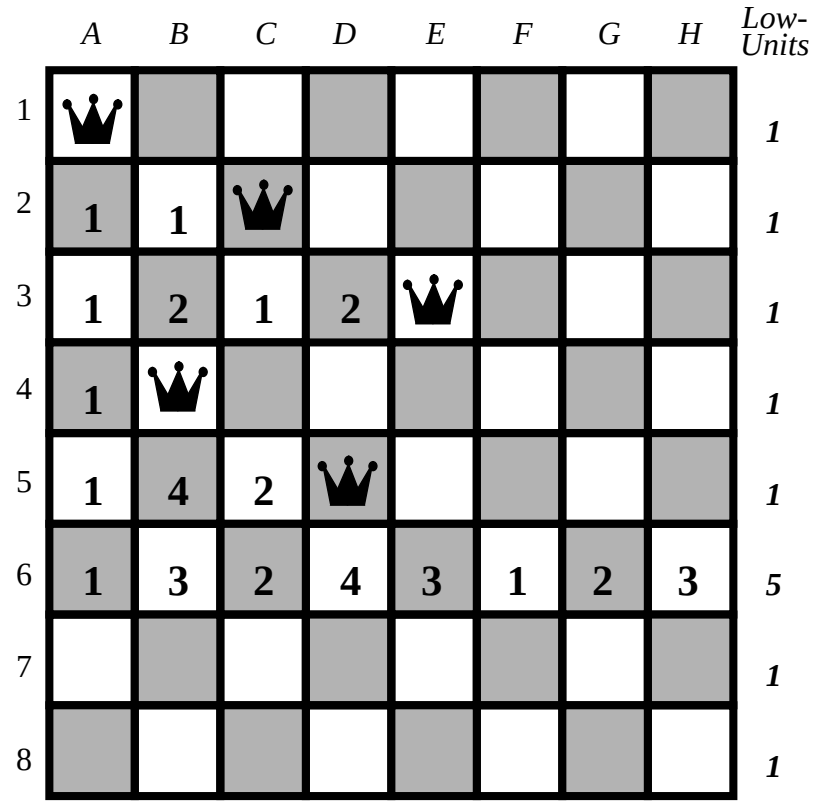


Figure 5.8 A board situation in the 8-queens problem showing the values of *Marks* and *LowUnits* at some state during Backmarking. The number in each square shows the value of *Mark* for that square. At this state, all the values of Queen 6 have been rejected, and consequently, *LowUnit*)6) has been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

5.5 Hybrid Algorithms and Truth Maintenance

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

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

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

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

5.6 Comparison of Algorithms

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

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

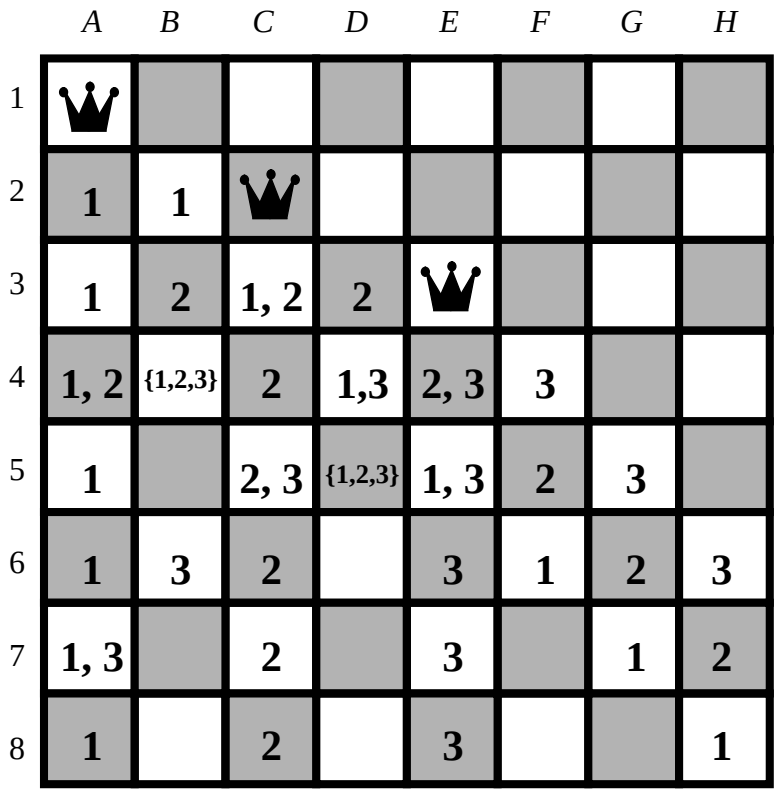


Figure 5.9 A board situation in the 8-queens problem for showing the role of Truth Maintenance in a DAC-L and DDBT hybrid algorithm. The number(s) in each square shows justification(s) for rejecting that square; e.g. the number 1 means the subject square is rejected because it is incompatible with the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

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

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

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

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

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

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

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

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

5.7 Summary

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

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

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

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

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

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

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

5.8 Bibliographical Remarks

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

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