Chapter 9
Solution synthesis

9.1 Introduction

As has been suggested in previous chapters, most research in CSP focuses on heuristic search and problem reduction. In this chapter, we shall look at techniques for constructively synthesizing solutions for CSPs.

We explained in previous chapters that problem reduction techniques are used to remove redundant values from variable domains and redundant compound labels from constraints, thus transforming the given problem to new ones which are hopefully easier to solve. Some problem reduction techniques, such as the adaptive consistency achievement algorithm, derive new constraints from the given problem. Problem reduction, in general, does not insist that all redundant compound labels are removed. The more effort one is prepared to spend, the more redundant compound labels one can hope to remove.

Solution synthesis techniques constructively generate legal compound labels rather than eliminating redundant labels or redundant compound labels. One can see solution synthesis as a special case of problem reduction in which the *n*-constraint for a problem with *n* variables is constructed, and all the *n*-compound labels which violate some constraints are removed. Alternatively, solution synthesis can be seen as "searching" multiple partial compound labels in parallel.

In this chapter, we shall introduce three solution synthesis algorithms, namely, Freuder's algorithm, Seidel's invasion algorithm and a class of algorithms called the Essex Algorithms. We shall identify situations in which these algorithms are applicable.

9.2 Freuder's Solution Synthesis Algorithm

The idea of solution synthesis in CSP was first introduced by Freuder. Freuder's algorithm is applicable to general CSPs in which one wants to find all the solutions. The basic idea in Freuder's algorithm is to incrementally build a lattice which represents the minimal problem (Definition 2-8). We call this lattice the **minimal problem graph**, or **MP-graph**. We use *MP-graph(P)* to denote the MP-graph of a CSP *P*.

Each node in the MP-graph represents a set of *k*-compound labels for *k* variables (note that this is different from a constraint graph which represents a CSP — there each node represents a variable). We call a node which contains *k*-compound labels a **node of order *k***, and use **order_of(Node)** to denote the order of Node. One node is constructed for each subset of variables in the CSP. So for a problem with *n* variables, 2^n nodes will be constructed. For convenience, we use **variables_of(*X*)** to denote the set of variables contained in the compound labels in the node *X* in an MP-graph. Further, we shall use **node_for(*S*)** to denote the node which represents the set of compound labels for the set of variables *S*. For example, if variables_of(*D*) = {*X*, *Y*}, then node *D* contains nothing but compound labels for the variables *X* and *Y*, such as {(<*X*,1><*Y*,*a*>), (<*X*,2><*Y*,*b*>), (<*X*,2><*Y*,*c*>)}. In this case, *D* = node_for({*X*, *Y*}) and order_of(*D*) is 2 (because *D* contains 2-compound labels).

Definition 9-1:

> A node *P* is a **minimal extension** of *Q* if *P* is of one order higher than node *Q*, and all the variables in variables_of(*Q*) are elements of variables_of(*P*). In other words, the variables of *P* are the variables of *Q* plus an extra variable (read *minimal_extension(P, Q)* as: *P* is a minimal extension of *Q*):
>
> $\forall$ csp(*P*): (*V*, *E*) = MP-graph(*P*):
> ($\forall$ *P*, *Q* $\in$ *V*:
> minimal_extension(*P*, *Q*) $\equiv$
> ((| variables_of(*P*) | = | variables_of(*Q*) | + 1) $\wedge$
> (variables_of(*Q*) $\subset$ variables_of(*P*))) ■

Obviously every node of order *k* is the minimal extension of *k* nodes of order *k* – 1. The arcs in the MP-graph represent constraints between the nodes. An arc exists between every node *P* of order *k* + 1 and every node *Q* of order *k* if and only if *P* is a minimal_extension of *Q*. See Figure 9.2 for the topology of an MP-graph.

9.2.1 Constraints propagation in Freuder's algorithm

The contents of each node D of order k in the MP-graph is determined by the following constraints, and the following constraints only:

(1) compound labels in D must satisfy the k-ary constraint on variables_of(D);

(2) upward propagation —
if a compound label cl is in D, then projections of cl must be present in every node of order $k-1$ which is connected to D. For example, if ($<x_1,v_1><x_2,v_2><x_3,v_3>$) is in node D, then ($<x_1,v_1><x_2,v_2>$) must be a member of the node for $\{x_1, x_2\}$; and

(3) downward propagation —
if a compound label cl is in D, then in every node of order $k+1$ which is connected to D there must be a compound label of which cl is a projection. For example, if ($<x_1,v_1><x_2,v_2><x_3,v_3>$) is in D, then at least one compound label ($<x_1,v_1><x_2,v_2><x_3,v_3><x_4,*>$) must be a member of the node for variables $\{x_1, x_2, x_3, x_4\}$, where '$*$' denotes a wildcard which represents any value that x_4 may take.

In other words, upward propagation attempts to eliminate compound labels in nodes of a higher order, and downward propagation attempts to eliminate compound labels in nodes of a lower order. To be exact, upward propagation and downward propagation achieve the properties **Upward_propagated** and **Downward_propagated**, which are defined below:

Definition 9-2 (Upward_propagated):

$$\begin{aligned}&\forall\ \text{csp}(P)\text{: } (V, E) = \text{MP-graph}(P)\text{:}\\ &\quad \text{Upward\_propagated}((V, E)) \equiv\\ &\qquad \forall\ Node_1, Node_2 \in V\text{:}\\ &\qquad\quad (\text{minimal\_extension}(Node_1, Node_2)\\ &\qquad\qquad \Rightarrow (\forall\ e_1 \in Node_1\text{: } (\exists\ e_2 \in Node_2\text{: projection}(e_1, e_2)))) \blacksquare\end{aligned}$$

Definition 9-3 (Downward_propagated):

$$\begin{aligned}&\forall\ \text{csp}(P)\text{: } (V, E) = \text{MP-graph}(P)\text{:}\\ &\quad \text{Downward\_propagated}((V, E))\text{:}\\ &\qquad \forall\ Node_1, Node_2 \in V\text{:}\\ &\qquad\quad (\text{minimal\_extension}(Node_1, Node_2)\\ &\qquad\qquad \Rightarrow (\forall\ e_2 \in Node_2\text{: } (\exists\ e_1 \in Node_1\text{: projection}(e_1, e_2)))) \blacksquare\end{aligned}$$

9.2.2 Algorithm Synthesis

The pseudo code for Freuder's solution synthesis algorithm (which we shall call Synthesis) is shown below:

```
PROCEDURE Synthesis(Z, D, C)
BEGIN
    /* Step 1: Initialization */
    V ← { }; E ← { };       /* the MP-graph of (Z, D, C) is (V, E) */
    FOR each x in Z DO
        BEGIN
            node_for({x}) ← { (<x,a>) | a ∈ D_x ∧ satisfies(<x,a>, C_x) };
            V ← V + {node_for({x})};
        END
    /* Step 2: Construction of higher-order nodes */
    FOR i = 2 to |Z| DO
        FOR each combination of i variables S in Z DO
            BEGIN
                IF (C_S ∈ C) THEN node_for(S) ← C_S;
                ELSE node_for(S) ← all possible combinations of labels
                    for S;
                V ← V + {node_for(S)};
                FOR each node X of which node_for(S) is a minimal
                    extension DO
                    BEGIN
                        E ← E + {(node_for(S), X)};
                        FOR each element cl of node_for(S) DO
                            IF (there exists no cl' in X such that projec-
                                tion(cl, cl')) holds
                            THEN node_for(S) ← node_for(S) – {cl};
                    END
                V ← Downward_Propagate(node_for(S), V);
            END
END /* of Synthesis */
```

Each node of order 1 is initialized to the set of all the values which satisfy the unary constraints of the subject variable. A node *N* of order *k* in general is constructed in the following way: If there exists any constraint on the variables_of(*N*), then the node_for(*N*) is instantiated to this constraint (readers are reminded that both the nodes and the constraints are treated as sets of compound labels). Otherwise, *N* is instantiated to the set of all possible combinations of values for the variables of *N*. Then *N* is connected to all the nodes of which *N* is the minimal extension.

After a node N is instantiated and linked to other nodes in the MP-graph, redundant compound labels in N are removed using the lower-order nodes which are adjacent to it. For example, the node for the variables $\{x_1, x_2, x_3, x_4\}$ is restricted by the nodes for the following sets of variables: $\{x_1, x_2, x_3\}$, $\{x_1, x_2, x_4\}$, $\{x_1, x_3, x_4\}$ and $\{x_2, x_3, x_4\}$. On the other hand, the content of N forms a constraint to all the nodes of a lower order, and such constraints are propagated using the Downward_Propagation procedure shown below. The Downward_Propagation and Upward_Propagation procedures call mutual recursively for as many times as necessary:

```
PROCEDURE Downward_Propagation(N, V)
/* propagate from node N to the set of nodes V in the MP-graph */
BEGIN
    FOR each node N' in V such that minimal_extension(N,N') DO
        BEGIN
            Original_N' ← N';
            FOR each element e' of N' DO
                IF (there exists no e in N such that projection(e,e'))
                THEN N' ← N' – {e'};
            IF (N' ≠ Original_N')
            THEN BEGIN
                            V ← Downward_Propagation(N', V);
                            V ← Upward_Propagation(N', V);
                        END
        END
    return(V);    /* content of the nodes in V may have been reduced */
END /* of Downward_Propagation */

PROCEDURE Upward_Propagation(N, V)
/* propagate from node N to the set of nodes V in the MP-graph */
BEGIN
    FOR each node N' in V such that minimal_extension(N',N) DO
        BEGIN
            Original_N' ← N';
            FOR each element e' of N' DO
                IF (there exists no e in N such that projection(e',e))
                THEN N' ← N' – {e'};
            IF (N' ≠ Original_N')
            THEN BEGIN
                            V ← Upward_Propagation(N', V);
                            V ← Downward_Propagation(N', V);
                        END
        END
    return(V);    /* content of the nodes in V may have been reduced */
END /* of Upward_Propagation */
```

Downward_Propagation(N, V) removes from every node N' of which N is a minimal_extension the compound labels which have no support from N. A compound label cl' in N' is supported by N if there exists a compound label cl in N such that cl' is a projection of cl. If the content of any N' is changed, the constraint must be propagated to all other nodes which are connected to N' through the calls to Downward_Propagation and Upward_Propagation.

Upward_Propagation(N, V) removes from every node N' which are minimal_extensions of N all the compound labels which do not have any projection in N. Similarly, if any N' is changed, the effect will be propagated to all other nodes connected to it.

Let us assume that a is the maximum size of the domains for the variables, and n is the number of variables in the problem. There are altogether ${}_nC_1 + {}_nC_2 + ... + {}_nC_n$ combinations of variables; hence there are 2^n nodes to be constructed in step 1. In the worst case, Upward_Propagation and Downward_Propagation remove only one compound label at a time. Since there are $O(a^n)$ compound labels, in the worst case, $O(a^n)$ calls of Upward_Propagation and Downward_Propagation are needed. In each call of Upward_Propagation, each element of every minimal_extension is examined. The number of elements in each minimal_extension is $O(a^n)$. Since there are $O(n)$ minimal_extensions, $O(na^n)$ projections have to be checked. Therefore, the worst case time complexity of Freuder's solution synthesis algorithm is $O(2^n + na^{2n})$. Since there are $O(2^n)$ nodes, and the size of each node is $O(a^n)$, the worst case space complexity of Synthesis is $O(2^n a^n)$.

9.2.3 Example of running Freuder's Algorithm

We shall use the 4-queens problem to illustrate Freuder's algorithm. The problem is to place four queens on a 4×4 chess board satisfying the constraints that no two queens can be on the same row, column or diagonal. To formulate it as a CSP, we use variables x_1, x_2, x_3 and x_4 to represent the four queens to be placed on the four rows of the board. Each of the variables can take a value from $\{A, B, C, D\}$ representing the four columns.

For convenience, we use subscripts to indicate the variables that each node represents: for example, N_{123} denotes the node for variables $\{x_1, x_2, x_3\}$. To start, the following nodes of order 1 will be constructed. Each node represents the set of values for the subject variable which satisfy the unary constraints:

N_1: {(A), (B), (C), (D)}
N_2: {(A), (B), (C), (D)}
N_3: {(A), (B), (C), (D)}
N_4: {(A), (B), (C), (D)}

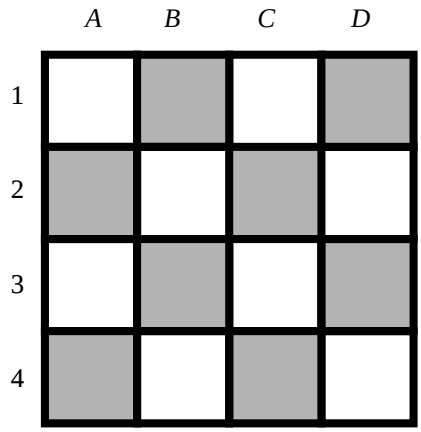


Figure 9.1 The board for the 4-queens problem

The binary constraints in the 4-queens problem determine the contents of the nodes of order 2 in the MP-graph. The following nodes of order 2 are initialized to the corresponding constraints:

N_{12}: {(*A*,*C*), (*A*,*D*), (*B*,*D*), (*C*,*A*), (*D*,*A*), (*D*,*B*)}
N_{13}: {(*A*,*B*), (*A*,*D*), (*B*,*A*), (*B*,*C*), (*C*,*B*), (*C*,*D*), (*D*,*A*), (*D*,*C*)}
N_{14}: {(*A*,*B*), (*A*,*C*), (*B*,*A*), (*B*,*C*), (*B*,*D*), (*C*,*A*), (*C*,*B*), (*C*,*D*), (*D*,*B*), (*D*,*C*)}
N_{23}: {(*A*,*C*), (*A*,*D*), (*B*,*D*), (*C*,*A*), (*D*,*A*), (*D*,*B*)}
N_{24}: {(*A*,*B*), (*A*,*D*), (*B*,*A*), (*B*,*C*), (*C*,*B*), (*C*,*D*), (*D*,*A*), (*D*,*C*)}
N_{34}: {(*A*,*C*), (*A*,*D*), (*B*,*D*), (*C*,*A*), (*D*,*A*), (*D*,*B*)}

After each node of order 2 is built, constraints are propagated downward to nodes of order 1. No change is caused by the propagation of these constraints. Next, the nodes of order 3 are built. For each combination of three variables, a node is constructed. Since no 3-constraint exists in the problem, all nodes N_{123}, N_{124}, N_{134} and N_{234} are instantiated to the cartesian product of the three domains: {(*A*,*A*,*A*), (*A*,*A*,*B*), ..., (*D*,*D*,*D*)}. Each of them is constrained by the relevant nodes of order 2. For example, N_{123} is restricted by N_{12}, N_{13} and N_{23}. Let '$*$' denote a wildcard. Since (*A*,*A*) is not a member of N_{12}, all the elements (*A*,*A*,$*$) are removed from N_{123}; since (*C*,*D*) is not a member of N_{23}, all the elements ($*$,*C*,*D*) are removed from N_{123}; and so on. After such local propagation of constraints, the nodes of order 3 are as follows:

N_{123}: {(*A*,*D*,*B*), (*B*,*D*,*A*), (*C*,*A*,*D*), (*D*,*A*,*C*)}
N_{124}: {(*A*,*C*,*B*), (*B*,*D*,*A*), (*B*,*D*,*C*), (*C*,*A*,*B*), (*C*,*A*,*D*), (*D*,*A*,*B*), (*D*,*B*,*C*)}
N_{134}: {(*A*,*D*,*B*), (*B*,*A*,*C*), (*B*,*A*,*D*), (*C*,*B*,*D*), (*C*,*D*,*A*), (*C*,*D*,*B*), (*D*,*A*,*C*)}
N_{234}: {(*A*,*D*,*B*), (*B*,*D*,*A*), (*C*,*A*,*D*), (*D*,*A*,*C*)}

To complete the construction of each node of order 3, Downward_Propagated is called by the Synthesis procedure. Since no (*A*,*C*,∗) and (*D*,*B*,∗) exist in any element of N_{123}, (*A*,*C*) and (*D*,*B*) are deleted from N_{12}. Similarly, since no (*A*,*D*,∗) exists in any of the compound labels of N_{124}, (*A*,*D*) must be deleted from N_{12} as well. As a result, N_{12} is reduced to:

N_{12} (updated): {(*B*,*D*), (*C*,*A*), (*D*,*A*)}

Similarly, other nodes of order 2 can be updated. After N_{12} is updated, constraints are propagated both downward and upward. Propagating downward, node N_1 is updated to {(*B*), (*C*), (*D*)}, because the value *A* does not appear in the first position (the position for x_1) of any element in node N_{12}. Similarly, node N_2 is updated to {(*A*), (*D*)}. Propagating upward from N_{12}, node N_{123} is updated to:

N_{123} (updated): {(*B*,*D*,*A*), (*C*,*A*,*D*), (*D*,*A*,*C*)}

Element (*A*,*D*,*B*) is discarded from N_{123} because (*A*,*D*) is no longer an element of N_{12}. Apart from N_{123}, all other nodes of order 3 in which are minimal_extensions of N_{12} have to be re-examined. For example, N_{124} will be updated to {(*B*,*D*,*A*), (*B*,*D*,*C*), (*C*,*A*,*B*), (*C*,*A*,*D*), (*D*,*A*,*B*)} (the element (*A*,*C*,*B*) is deleted from N_{124} because (*A*,*C*) in no longer a member of N_{12}).

The result of N_{123} being restricted can again be propagated downward to all the nodes of order 2 which are nodes for subsets of {x_1, x_2, x_3}. For example, N_{13} will be restricted to:

N_{13} (updated): {(*B*,*A*), (*C*,*D*), (*D*,*C*)}

because only these elements are accepted by elements of the updated N_{123}. The propagation process will stop when and only when no more nodes are updated. Finally, the following node of order 4 will be constructed using all the nodes of order 3:

N_{1234}: {(*B*,*D*,*A*,*C*), (*C*,*A*,*D*,*B*)}

Node N_{1234} contains the only two possible solutions for this problem. Figure 9.2 shows the final MP-graph for the 4-queens problem built by Freuder's algorithm. Every compound label in every node appears in at least one solution tuple.

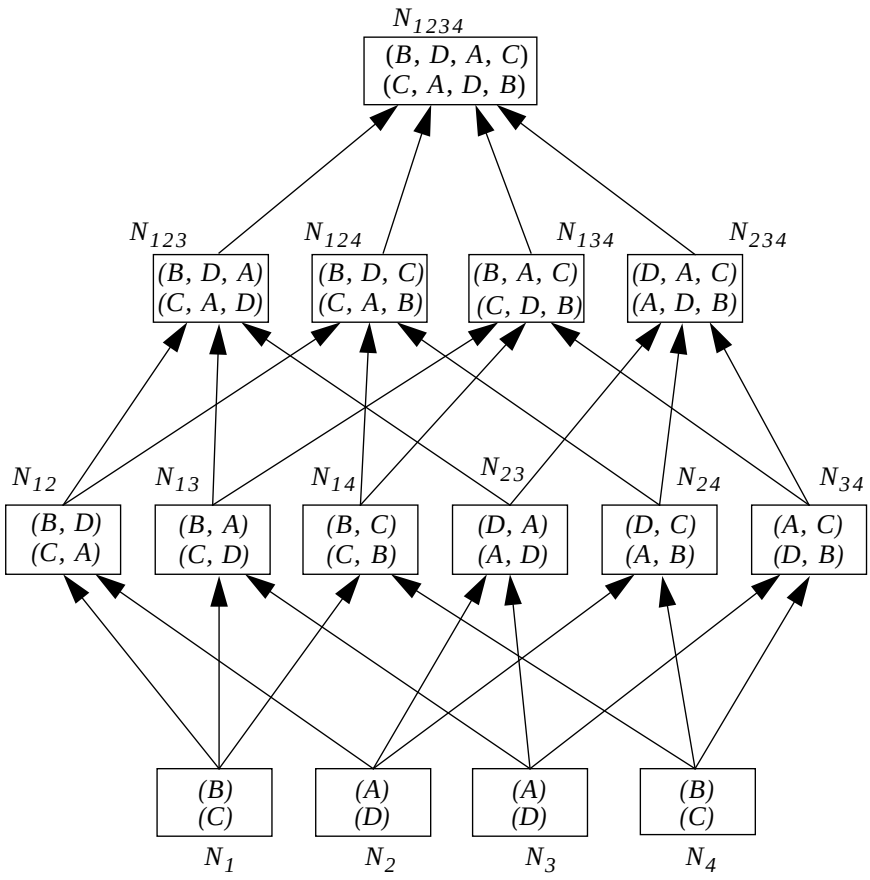


Figure 9.2 The MP-graph constructed by Freuder's algorithm in solving the 4-queens problem (after propagation of all the constraints)

9.2.4 Implementation of Freuder's synthesis algorithm

Program 9.1, *synthesis.plg*, shows a Prolog implementation of the above Synthesis procedure for tackling the *N*-queens problem. In this program, the nodes of the MP-graph and their contents are asserted into the Prolog database. Since 2^n nodes must be built for a problem of *n* variables, and constraints are propagated through the network extensively, carrying the nodes as parameters would be too expensive and clumsy.

For each node *N* that has been built, node(*N*) is asserted in *synthesis.plg*. *N* is simply a list of numbers which represent the rows. Node(*N*) is checked before constraint is propagated to or from it. If node(*N*) has not been built yet, then no constraint is propagated to and from it. If node(*N*) is already built, but no compound label is stored in it, then we know that there exists no solution to the problem.

Each compound label *cl* which is considered to be legal is asserted in a predicate content(*cl*). Clauses in the form of content/1 could be retracted in constraint propagation. For clarity, Program 9.1 reports the progress of the constraint propagation.

9.3 Seidel's Invasion Algorithm

The **invasion algorithm** is used to find all solutions for binary CSPs. Although it is possible to extend it to handle general CSPs, using it for solving CSPs which have *k*-ary constraints for large *k* would be inefficient. The invasion algorithm exploits the topology of the constraint graph, and is especially useful for problems in which every variable is involved in only a few constraints. Basically, it orders the variables and constructs a directed graph where each node represents a legal compound label and each arc represents a legal extension of a compound label. The variables are processed one at a time. When each variable is processed, the invasion algorithm generates nodes that represent the compound labels (or partial solutions) which involve this variable. After all the variables have been processed, each complete path from the last node to the first node in the graph represents a legal solution tuple.

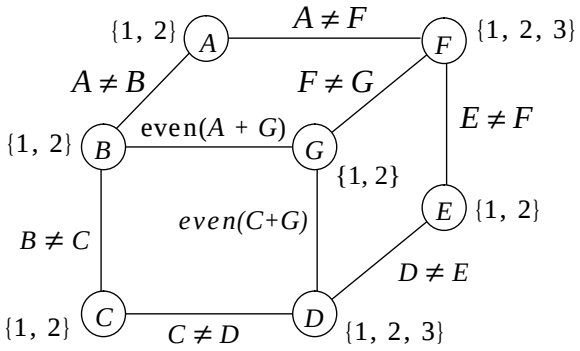
9.3.1 Definitions and Data Structure

Definition 9-4:

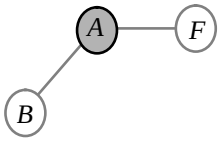
Given a graph G of n nodes and a total ordering $<$ on its nodes, an **invasion** is a sequence of partial graphs (Definition 3-27) $G_1, G_2, ..., G_n$ with the first 1, 2, ..., n nodes under the ordering $<$:

$$\forall \text{ graph}((V, E)): (\forall <: \text{total\_ordering}(V, <): |V| = n:$$
$$(\text{invasion}((G_1, G_2, ..., G_n), (V, E), <) \equiv$$
$$(\forall i: 1 \le i < n:$$
$$((G_i = (V_i, E_i) \wedge G_{i+1} = (V_{i+1}, E_{i+1})) \Rightarrow$$
$$(\text{partial\_graph}(G_i, G_{i+1}) \wedge$$
$$\exists y \in V_{i+1}: (V_{i+1} = V_i + \{y\} \wedge \forall x \in V_i: x < y))))))$$ ■

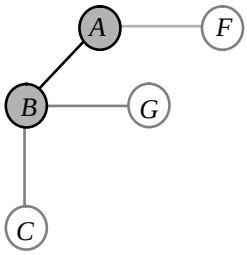
In other words, the partial graph G_i in an invasion consists of the first i nodes of V according to the ordering of the invasion. Figure 9.3 shows a constraint graph and a possible invasion.



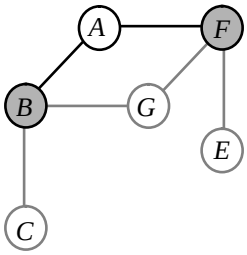
(a) The constraint graph to be labelled



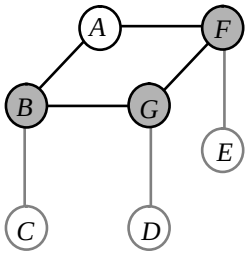
(b) Node *A* is invaded



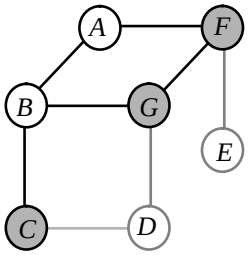
(c) Node *B* is invaded



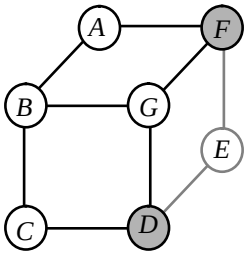
(d) Node *F* is invaded



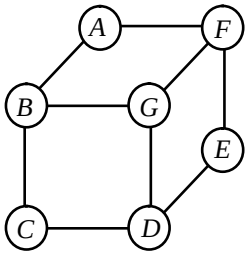
(e) Node *G* is invaded



(f) Node *C* is invaded



(g) Node *D* is invaded



(h) Node *E* is invaded

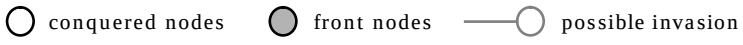


Figure 9.3 Example of an invasion

The invasion algorithm is very similar to the Find_Minimal_Bandwidth algorithm described in Chapter 6, and therefore we shall refer to the definitions there. By Definition 6-4, the nodes in the partial graph G_i in an invasion and its ordering is a *partial layout* $(V, <)$. Here, we shall use the terms *conquered nodes* and *active nodes* as they were defined in Chapter 6 (Definitions 6-6 and 6-7).

Definition 9-5:

The **front** of an invasion graph is the set of active nodes under the ordering of the invasion:

$$\forall\ \text{graph}((V, E)): \forall\ <: \text{total\_ordering}(Z, <):$$
$$\forall\ (V_1, E_1), ..., (V_n, E_n): \text{invasion}(((V_1, E_1), ..., (V_n, E_n)), (V, E), <):$$
$$(\forall\ i: 1 \le i \le |\ V\ | : (\text{front}((V_i, E_i)) \equiv$$
$$\{v \mid \text{active\_node}(v, (V_i, <), (V, E))\}))))\ \blacksquare$$

Given a CSP P, the invasion algorithm maintains a directed graph, which we shall call the **solution graph**, which records the set of all partial solutions. Let S be a solution graph and $S = (V_S, E_S)$. Each node in V_S represents a compound label for the variables in front(G_i) for some i. There are two special nodes: a start node and an end node. The start node represents the 0-compound label and the end node represents the compound labels for the variables in front(G(P)) (which is also empty because G(P) has no active nodes). Each arc in E_S goes from a compound label for front(G_{i+1}) to a compound label for front(G_i). The arcs are marked by a possible value: the arc between front(G_i) and front(G_{i+1}) is marked by a value in the domain of the i-th variable in the ordering. When the algorithm terminates, each path from the end node to the start node represents a solution. See Figure 9.4 later for the topology of a solution graph.

9.3.2 The invasion algorithm

The basic idea of the invasion algorithm is to look at the partial graphs of the invasion according to the given ordering, and augment the solution graph in the following way: for each compound label *cl* for the variables of front(G_i), and for each value v in the domain of the $(i + 1)$-th variable, check whether *cl* is compatible with v. If so, then create a node N for the variables of front(G_{i+1}) if such node does not already exist. Then create an arc from N to the node which represents *cl*.

```
PROCEDURE Invasion(Z, D, C, <)
BEGIN
   /* Si is the set of nodes for the i-th partial graph in the invasion */
```

```
/* create the start node which represents the 0-compound label */
S0 ← {( )};
FOR i = 1 to |Z| DO
    BEGIN
        Gi ← i-th partial graph in the invasion of the graph G(Z, D,
            C) according to <;
        Si ← { };
        FOR each CL in Si-1 DO
            FOR each value v in domain xi DO
                IF (CL + <xi,v> satisfies CE(variables_of(CL) + {xi}))
                THEN BEGIN
                            CL' ← CL + <xi,v> – labels for conquered
                                nodes in Gi;
                            Si ← Si + {CL'};
                            create arc from CL' to CL and mark it with
                                <xi,v>;
                        END
        IF (Si = { }) THEN report no solution;
    END
END /* of Invasion */
```

The nodes in the solution graph are logically grouped into sets: S_i is the set of nodes for the i-th partial graph in the invasion. Readers are reminded that CE(S) is the constraint expression of a set of variables S (Definition 2-9). *CL'* represents the compound label of the variables in front(G_i). If n is the number of variables in the CSP, then G_n contains just one node, which we call the end node (this is because the front of G_i is by definition an empty set). The Invasion algorithm constructs S_0, S_1, ..., S_n in that order. After termination of Invasion, the solution graph comprises the sets of nodes in $S_0 + S_1 + ... + S_n$. Each path from the end to the start node represents a solution to the CSP. If any set S_i is found to be empty after the i-th partial graph has been processed, then no solution exists for the input CSP.

9.3.3 Complexity of invasion and minimal bandwidth ordering

Let n be the number of variables, a the maximum domain size, and e the number of constraints in a binary CSP. Further let f be the maximum size of front(G_i) for all $1 \le i \le n$. In the following we show that the time and space complexity of the invasion algorithm are $O(ea^{f+1})$ and $O(na^{f+1})$, respectively.

Since f is the maximum size of front(G_i) for all i, there are at most a^f f-compound

labels *CL* in S_{i-1}. Therefore, when a value *v* in x_i is being processed in the inner FOR loop, $<x_i,v>$ is checked against at most a^f labels in *CL*. At most a^{f+1} compatibility checks are required to process each *CL*, hence at most fa^{f+1} checks are required to process each S_{i-1}. If each compatibility check between every pair of labels takes a constant time, then the time complexity of the algorithm is $O(nfa^{f+1})$. But since there are no more than *e* constraints, $n \times f$ is bounded by *e*. So the time complexity of the algorithm is $O(ea^{f+1})$.

Since there are at most a^f *f*-compound labels at the front of a partial graph, there are at most a^f nodes in S_i for all *i*. So there are at most na^f nodes in the solution graph *S*. Each *f*-compound label in the nodes of S_{i-1} is compatible with at most all *a* values in x_i. Therefore, no more than a^{f+1} arcs go from S_{i-1} to S_i. If each node is stored in a constant space, then the space complexity of the invasion algorithm is dominated by $O(na^{f+1})$.

The above analysis shows that the value *f*, i.e. the maximum front size, significantly affects the complexity of the invasion algorithm. The natural question then is how to find an invasion which *f* is minimal. In Chapter 6, we introduced the concept of bandwidth and an algorithm for finding the minimal bandwidth. Since the front is defined as the set of active nodes, an ordering which has the minimal bandwidth has the smallest maximum front size *f*. Therefore, the time and space complexity of the invasion algorithm are $O(ea^{b+1})$ and $O(na^{b+1})$, where *b* is the (minimal) bandwidth of the graph.

In the discussion of bandwidth in Chapter 6, we said that the time complexity of finding the bandwidth of a graph is $O(n^b)$, and any CSP whose constraint graph's bandwidth is no larger than *b* can be solved in time $O(n^b+a^{b+1})$ and space $O(n^b+a^b)$. In the case when a^{b+1} dominates the time complexity, this result is consistent with our analysis of the complexity of the invasion algorithm.

Seidel claims that it is possible to extend the invasion algorithm to non-binary CSPs. This can be done by modifying the definition of connectivity appropriately. However, one must note that when non-binary constraints are considered, the time complexity of the algorithm is changed. When the compatibility between *CL* and $<x_i,v>$ is checked, more than *f* checks could be needed if the constraints are not limited to binary. In the worst case, there could be 2^f tests. When this is the case, the time complexity of the algorithm would become $O(ea^{2^f+1})$ instead of $O(ea^{f+1})$.

9.3.4 Example illustrating the invasion algorithm

The following example from Seidel [1981] illustrates the invasion algorithm. Suppose there are four integer variables, x_1, x_2, x_3, and x_4, and the domains for all of them are the same: {1, 2, 3}. Let the following be the only constraints in the problem:

$$x_1 < x_2$$
$$x_1 < x_3$$
$$x_2 \leq x_4$$
$$x_3 \leq x_4$$

The problem is to find all combinations of assignments to the four variables, satisfying all the constraints. The constraint graph of this problem is shown in Figure 9.4(a). Suppose the (arbitrary) ordering of the invasion is (x_1, x_2, x_3, x_4).

Figure 9.4(b) shows the solution graph generated by the invasion algorithm. G_1 contains x_1 only, which is connected to uninvaded nodes. So the front of G_1 is $\{x_1\}$. Since all the possible labels of x_1 are compatible with the 0-compound label (), nodes for all (<x_1,1>), (<x_1,2>) and (<x_1,3>) are created and put into S_1. G_2 contains x_1 and x_2. Since both of them are connected to some uninvaded nodes, both are in the front of G_2. Since both <x_2,2> and <x_2,3> are compatible with <x_1,1>, nodes for both (<x_1,1><x_2,2>) and (<x_1,1><x_2,3>) are created. Node (<x_1,2><x_2,3>) is created because <x_1,2> is compatible with <x_2,3>. G_3 contains x_1, x_2 and x_3, among which x_1 is conquered. So the front of G_3 is $\{x_2, x_3\}$ and nodes for S_3 are 2-compound labels for x_2 and x_3.

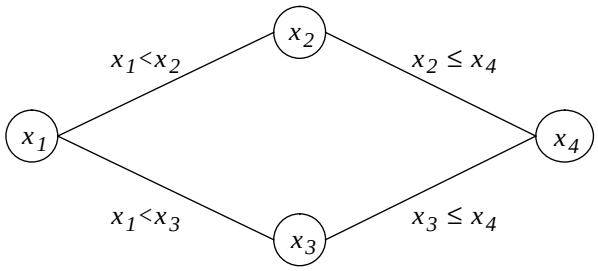
The solutions can be found following the paths from the end node to the start node. An example of two solutions shown in the solution graph are:

(<x_1,1><x_2,2><x_3,2><x_4,2>)

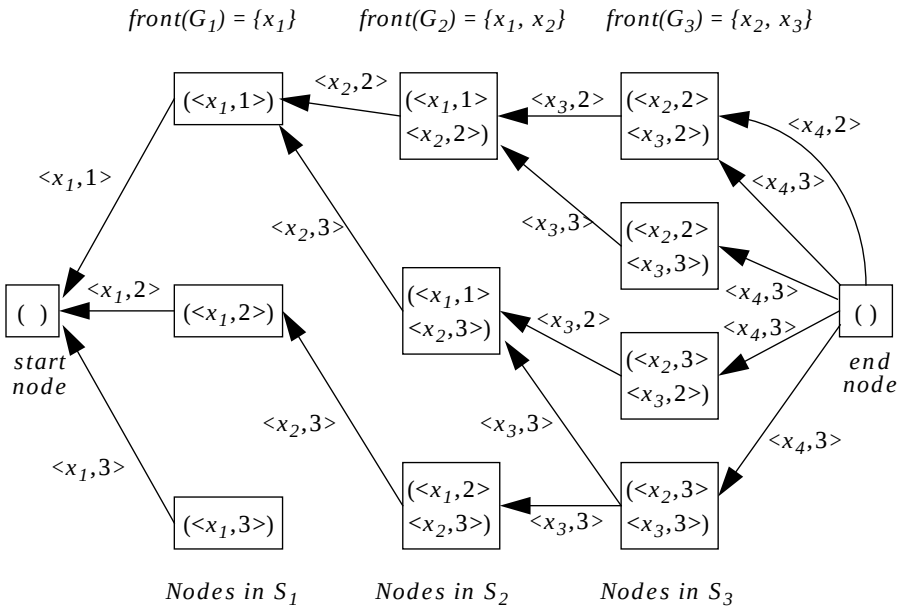
and (<x_1,1><x_2,2><x_3,3><x_4,3>).

9.3.5 Implementation of the invasion algorithm

Program 9.2, *invasion.plg*, shows an implementation of the invasion algorithm. It is applicable to binary constraint problems only, though it is quite easy to modify it to handle general problems (one needs to modify satisfy_constraints/2 and find_new_-front/3 in update_sg_aux/6). It assumes a particular form of the input data, and therefore has to be modified if the input is in a different format. The example given at the beginning of the program (under the heading *Notes*) is the same example as that in the preceding section, with variable names changed from x_1, x_2, x_3 and x_4 to *w*, *x*, *y* and *z*.



(a) The constraint graph an example CSP (all domains are {1, 2, 3})



(b) The solution graph generated by the invasion algorithm, assuming that the ordering for invasion is (x_1, x_2, x_3, x_4); each path from the end node to the start node represents a solution

Figure 9.4 Example showing the output of the invasion algorithm (one solution is (<x_1,1><x_2,2><x_3,2><x_4,2>))

9.4 The Essex Solution Synthesis Algorithms

In this section, we shall introduce a class of solution synthesis algorithms that were developed with the intention of exploiting advanced hardware. These algorithms are inspired by and modifications of Freuder's algorithm in Section 9.2. They are applicable to general problems, though particularly useful for binary constraint problems. The possible exploitation of hardware by these algorithms will be discussed in Section 9.5.

9.4.1 The AB algorithm

The basic Essex solution synthesis algorithm is called **AB** (which stands for Algorithm Basic). As Freuder's algorithm, AB synthesizes solution tuples by building a graph in which each node represents a set of compound labels for a particular set of variables. We shall call the graphs generated by AB **AB-Graphs**. As before, we shall use variables_of(N) to denote the set of variables for the node N in the AB-graph, and order_of(N) to denote its order. Unlike in Freuder's algorithm, nodes in an AB-Graph are partially ordered, and only adjacent nodes are used to construct nodes of higher order. The ordering and adjacency of the nodes are defined as follows.

Definition 9-6 (ordering of nodes in AB):

Given any total ordering of the variables in a CSP, the nodes of order 1 in the AB-graph are ordered according to the ordering of the variables that they represent. The **ordering of nodes** of higher order is defined recursively. For all nodes *P* and *Q* of the same order, *P* is before node *Q* if there exists a variable in variables_of(*P*) which is before all the variables in variables_of(*Q*):

$\forall$ csp(P): (V, E) = AB-graph(P):
  ($\forall$ <: total_ordering({$N \mid N \in V \wedge$ order_of(N) = 1}, <):
    ($\forall$ P, $Q \in V$: order_of(P) = order_of(Q)) $\wedge$ order_of(P) > 1:
      ($P < Q \equiv \exists\, x \in$ variables_of(P): $\forall\, y \in$ variables_of(Q): $x < y$))) ■

Definition 9-7 (adjacency of nodes in AB):

Two nodes of the same order are **adjacent** to each other if and only if one of them is before the other, and there exists no node of the same order which is between them in the ordering:

$\forall$ csp(P): (V, E) = AB-graph(P):
  ($\forall$ <: total_ordering({$N \mid N \in V \wedge$ order_of(N) = 1}, <):
    ($\forall$ P, $Q \in V$: order_of(P) = order_of(Q):
      (adjacent_ordered_node(P, Q, <) $\equiv$

$$((P < Q \wedge \neg \exists \text{ node } R: \\ (\text{order\_of}(R) = \text{order\_of}(Q) \wedge P < R \wedge R < Q)) \vee \\ (Q < P \wedge \neg \exists \text{ node } R: \\ (\text{order\_of}(R) = \text{order\_of}(Q) \wedge Q < R \wedge R < P))))) \blacksquare$$

Since only adjacent nodes are used to construct new nodes, the AB-graph that AB generates is actually a tangled binary tree. This tree will be constructed from the tips to the root, with n, $n-1$, $n-2$, ..., 3, 2, 1 nodes being constructed for each order, where n is the number of variables in the problem. The root of this tree is the node for solution tuples (see Figure 9.6 for the topology of the AB-graph). The pseudo code for the algorithm AB is shown below.

```
PROCEDURE AB(Z, D, C)
/* Z: a set of variables, D: index to domains, C: a set of constraints */
BEGIN
    /* initialization */
    give the variables an arbitrary ordering <;
    S ← { }      /* S = set of nodes in the AB-Graph to be constructed */
    FOR each variable x in Z DO
        S ← S ∪ { (<x,v>) | v ∈ D_x ∧ <x,v> ∈ C_x };
    k = 1;
    /* synthesis of solutions */
    WHILE (k ≤ |Z|) DO
        BEGIN
            FOR each pair of adjacent nodes P, Q in S such that P < Q
                DO S ← S ∪ {Compose(P,Q)};
            k = k + 1;
        END
    return node of order |Z| in S which represents the set of all solu-
        tion tuples;
END /* of AB */
```

The node of order $|Z|$ contains all the solution tuples for the problem (this node could be an empty set). AB ensures that in Compose(P, Q), P and Q are nodes of the same order, adjacent to each other and $P < Q$. This implies that the sets variables_of(P) and variables_or(Q) differ in exactly one element, and P's unique element is before all of Q's elements. In the procedure Compose, we assume that:

$$\text{variables\_of}(P) = \{x\} + W \\ \text{variables\_of}(Q) = W + \{y\}$$

where W is a set of variables and $x < y$. Following we show the Compose procedure for binary CSPs:

```
PROCEDURE Compose(P, Q)
BEGIN
   R ← { }; /* node to be returned */
   FOR each element (<x,a><x1,v1>...<xm,vm>) in P
      FOR each element (<x1,v1>...<xm,vm><y,b>) in Q
         IF satisfies((<x,a><y,b>), Cx,y)
         /* only binary constraints are checked here; in dealing with
            general CSPs, check if satisfies((<x,a><x1,v1> ...
            <xm,vm><y,b>), CE({x,x1,...,xm,y}) holds */
         THEN R ← R + {(<x,a><x1,v1>...<xm,vm><y,b>)};
   return(R);
END /* of Compose */
```

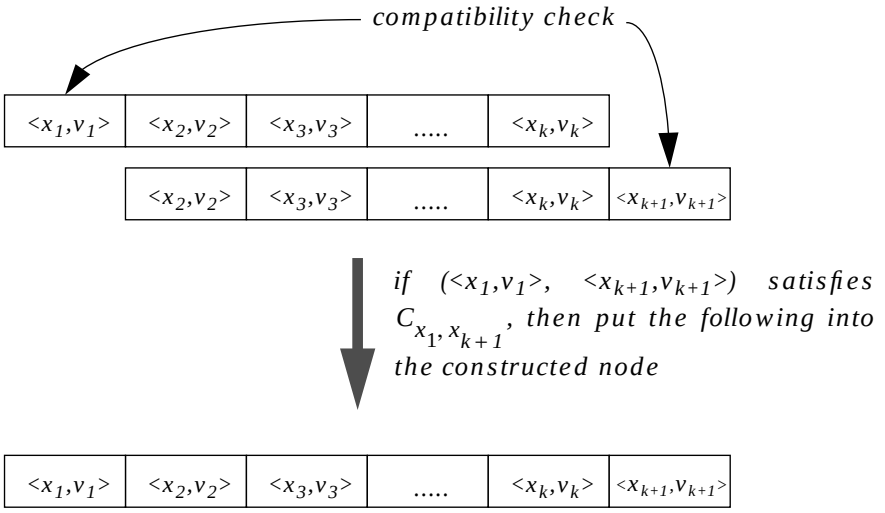
The procedure Compose(P ,Q) picks from the two given nodes P and Q a pair of elements which have the same projection to the common variables, and checks to see if the unique labels for the differing variables are compatible. If they are, a compound label containing the union of all the labels is included in the node to be returned.

For general CSPs, Compose has to check whether the differing variables are involved in any general constraints which might involve the common variables. If such constraint exists, Compose has to check whether the combined compound label satisfies all such constraints before it is put into the constructed node. Figure 9.5 summarizes the constraints being checked in Compose for both binary constraint problems and general problems.

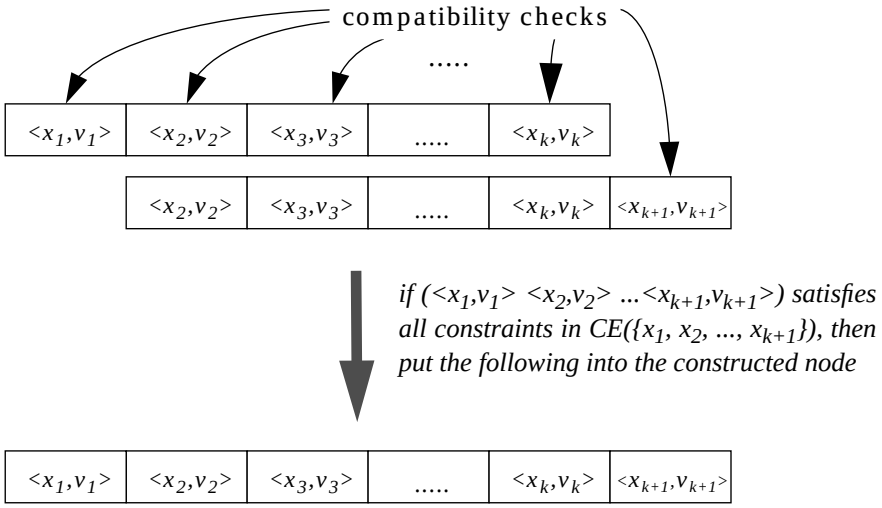
There are $(n - k + 1)$ nodes of order k in the AB-graph. When each of these nodes is constructed, two nodes of order $k - 1$ will be passed as parameters to Compose. The time complexity of Compose is determined by the size of these input nodes. The size of a node of order $k - 1$ is $O(a^{k-1})$ in the worst case. Compose has to consider each combination of two elements from the two input nodes. Therefore, the time complexity of Compose is $O(a^{2k-2})$. So the time complexity of AB is $\sum_{k=1}^{n} (n - k + 1)\, a^{2k-2}$, which is dominated by the term where $k = n$, i.e. $O(a^{2n-2})$. The largest possible node created by AB has the size a^n. Therefore, the worst case space complexity of AB is $O(a^n)$. The memory requirement of AB will be studied in greater detail in Section 9.5.1.

9.4.2 Implementation of AB

Program 9.3, *ab.plg*, is a Prolog implementation of the AB algorithm for solving the N-queens problem. A node is represented by:



(a) Compose for binary constraints problems



(b) Compose for general CSPs

Figure 9.5 Constraints being checked in the Compose procedure

$$[X_1, X_2, ...]\text{-}[V_1, V_2, ...]$$

in the program, where $[X_1, X_2, ...]$ is the list of variables for the subject node, and each of V_i's is a value to be assigned to the variable X_i. Syn/2 is given the list of all nodes of order 1. In each of its recursive calls, it will generate the set of nodes of one order higher (through calling syn_nodes_of_current_order/2), until either the solutions are generated, or it is provable that no solution exists.

9.4.3 Variations of AB

The efficiency of AB can be improved in the initialization. AB can also be modified should constraint propagation be worthwhile (as in Freuder's algorithm). These variations of AB are described briefly in the following sections.

9.4.3.1 Initializing AB using the MBO

The nodes are ordered arbitrarily in AB, but the efficiency of AB could be improved by giving the nodes certain ordering. The observation is that the smaller the nodes, the less computation is required for composing the higher order nodes. Although the size of the nodes of order 1 are determined by the problem specification, the sizes of the higher order nodes are determined by how much the variables of those nodes constrain each other. When the tightness of individual constraints are easily computable, one may benefit from putting the tightly constrained variables closer together in the ordering of the nodes of order 1 in AB. One heuristic is to give the variables a minimal bandwidth ordering (MBO) during initialization. For algorithms for finding the minimal bandwidth ordering and their implementations, readers are referred to Section 6.2.2 in Chapter 6.

9.4.3.2 The AP algorithm

Constraints are not propagated upward or downward in AB as they are in Freuder's algorithm. This is because AB is designed to exploit parallelism. All the nodes of order k are assumed to be constructed simultaneously. Propagating constraints will reduce the nodes' sizes and reduce the number of compatibility checks, but hamper parallelism. This is because, as Kasif [1990] has pointed out, consistency achievement is sequential by nature.

Constraints could be fully or partially propagated in AB if desired. The AP algorithm (P for Propagation) is a modification of AB in that constraints are partially propagated. In AP, if nodes P and Q are used to construct R, and $P < Q$, then constraints are propagated from R to Q (which will be used to construct the next node of one order higher than P and Q). Constraints are not propagated from R to P, or from Q to nodes of a lower order.

It is possible to extend AP further to maintain Upward_propagated (Definition 9-2) and Downward_propagated (Definition 9-3). Doing so could reduce the size of the nodes, at the cost of more computation. Whether it is justifiable to do so depends on the application. The decision on how much propagation to perform in AP is akin to the decision on what level of consistency to achieve in problem reduction. Program 9.4, *ap.plg*, is a Prolog implementation of the AP algorithm for solving the *N*-queens problem.

9.4.4 Example of running AB

We shall use the 4-queens problem to illustrate the AB procedure. As before, we shall use one variable to represent the queen in one row, and call the four variables x_1, x_2, x_3 and x_4. We shall continue to use subscripts to indicate the variables that each node represents: for example, N_{123} denotes the node for variables $\{x_1, x_2, x_3\}$.

Since all the variables x_1, x_2, x_3 and x_4 can take values *A*, *B*, *C* and *D*, all nodes of order 1 are identical:

N_1: {(*A*), (*B*), (*C*), (*D*)}
N_2: {(*A*), (*B*), (*C*), (*D*)}
N_3: {(*A*), (*B*), (*C*), (*D*)}
N_4: {(*A*), (*B*), (*C*), (*D*)}

From the nodes of order 1, nodes of order 2 are constructed:

N_{12}: {(*A*,*C*), (*A*,*D*), (*B*,*D*), (*C*,*A*), (*D*,*A*), (*D*,*B*)}
N_{23}: {(*A*,*C*), (*A*,*D*), (*B*,*D*), (*C*,*A*), (*D*,*A*), (*D*,*B*)}
N_{34}: {(*A*,*C*), (*A*,*D*), (*B*,*D*), (*C*,*A*), (*D*,*A*), (*D*,*B*)}

As described in the algorithm, only adjacent nodes of order 1 are used to construct nodes of order 2. So nodes such as N_{13} and N_{24} will not be constructed. Node N_{12} suggests that compound labels (<x_1,*A*><x_2,*C*>), (<x_1,*A*><x_2,*D*>), (<x_1,*B*><x_2,*D*>), etc. are all legal compound labels, as far as the constraint C_{x_1, x_2} is concerned. With these nodes of order 2, the following nodes of order 3 will be generated:

N_{123}: {(*A*,*D*,*B*), (*B*,*D*,*A*), (*C*,*A*,*D*), (*D*,*A*,*C*)}
N_{234}: {(*A*,*D*,*B*), (*B*,*D*,*A*), (*C*,*A*,*D*), (*D*,*A*,*C*)}

Finally, the following node of order 4 will be generated, which contains all the solution for the problem:

N_{1234}: {(*B*,*D*,*A*,*C*), (*C*,*A*,*D*,*B*)}

Node N_{1234} contains two compound labels:

($<x_1,B><x_2,D><x_3,A><x_4,C>$)
and ($<x_1,C><x_2,A><x_3,D><x_4,B>$)

are the only two solutions for this problem. At this stage, the AB-graph (which is a tangled binary tree) in Figure 9.6 is constructed.

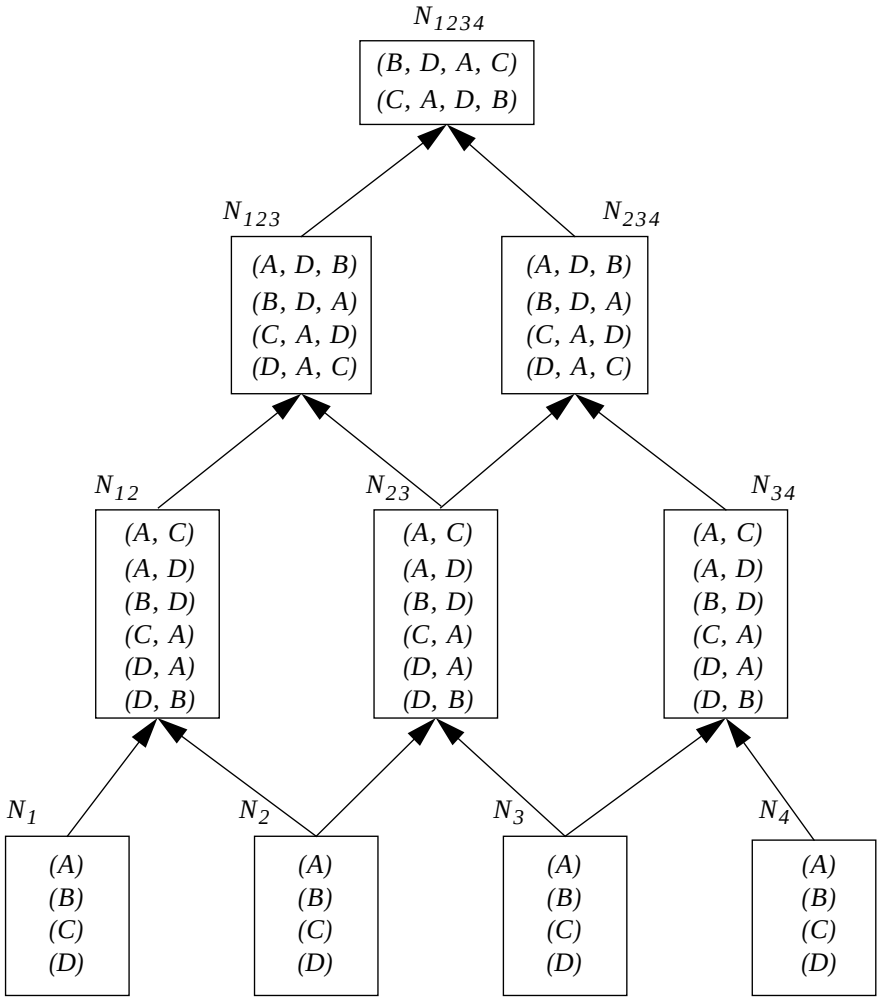


Figure 9.6 The tangled binary tree (AB-graph) constructed by the AB algorithm in solving the 4-queens problem

9.4.5 Example of running AP

For the simple example shown in the last section, different nodes will be generated by AP. Before constraints are propagated, the nodes of order 1 and order 2 in running AP are exactly the same as those in AB. After the following node N_{123} is constructed:

N_{123}: {(*A*,*D*,*B*), (*B*,*D*,*A*), (*C*,*A*,*D*), (*D*,*A*,*C*)},

N_{123} will form a constraint to node N_{23} (but not N_{12} because it will not be used to construct any more nodes). (*B*,*D*) will be removed from N_{23} because there is no (∗,*B*,*D*) in N_{123} (where '∗' represents a wildcard). Similarly (*C*,*A*) will be removed from N_{23} because there is no (∗,*C*,*A*) in N_{123}. So the node N_{23} is updated to:

N_{23} (updated): {(*A*,*C*), (*A*,*D*), (*D*,*A*), (*D*,*B*)}

The updated N_{23} will be used to build N_{234}:

N_{234}: {(*A*,*D*,*B*), (*D*,*A*,*C*)}

Finally, the node of order 4 where solutions are stored is constructed:

N_{1234} (solution): {(*B*,*D*,*A*,*C*), (*C*,*A*,*D*,*B*)}

9.5 When to Synthesize Solutions

In this section, we shall firstly identify the types of problems which are suitable for solution synthesis. Then we shall argue that advanced hardware could make solution synthesis more attractive than in the past.

9.5.1 Expected memory requirement of AB

Since solution synthesis methods are memory demanding by nature, we shall examine the memory requirements for AB in this section. Given any CSP, one can show that the size of the nodes in AB grows at a decreasing rate as the order of the node grows.

In a problem with n variables, n nodes of order 1 will be created. Among the n nodes of order 1, there are $n - 1$ pairs of adjacent nodes. Therefore, $n - 1$ nodes of order 2 will be constructed. There would be $n - 2$ nodes of order 3, $n - 3$ nodes of order 4, ..., and 1 node of order n. The total number of nodes in the binary tree is $n(n + 1) / 2$. Therefore, the number of nodes is $O(n^2)$. The complexity of composing a new node is $O(s_1 \times s_2)$, where s_1 and s_2 are the sizes of the two nodes used to construct the new node.

The size of the nodes is determined by the tightness of the problem: the tighter the constraints, the smaller the sizes of the nodes. For simplicity, we shall limit our analysis to binary constraint problems here. Let r be the proportion of binary-compound labels which are allowed in each binary constraint. Assume for simplicity that the domain size of every variable in the CSP is a. The expected size of a node of order 2 is $a \times a \times r$. Given two labels $<x_1, v_1>$ and $<x_2, v_2>$ the chance of a label $<x_3, v_3>$ being compatible with them simultaneously is r^2. The chance of a label $<x_4, v_4>$ being compatible with all $<x_1, v_1>$, $<x_1, v_2>$ and $<x_1, v_3>$ is r^3.

In general, the size of a node of order k, *S(k)*, is:

$$S(k) = r \times r^2 \times r^3 \times ... \times r^{k-1} \times r^k \times a^k = r^{k(k-1)/2} \times a^k$$

Since $0 \le r \le 1$, $r^{k(k-1)/2}$ should decrease at a faster rate than a^k increases. We can find k which has the maximum number of elements in its nodes by finding the derivative of *S(k)* and making it equal to zero. Let $t = \sqrt{r}$:

$$\frac{d(S(k))}{d(k)} = \frac{d(t^{k(k-1)})}{d(k)} \times a^k + \frac{d(a^k)}{d(k)} \times t^{k(k-1)}$$
$$= (2k-1) \times ln(t) \times t^{k(k-1)} \times a^k + ln(a) \times a^k \times t^{k(k-1)}$$

If $\frac{d(S(k))}{d(k)} = 0$, then we have: $(2k-1) \times \ln(t) + \ln(a) = 0$. Therefore, $k = \frac{1 - ln(a)/ln(t)}{2}$, or $k = \frac{1 - 2 \times ln(a)/ln(r)}{2}$, which is the order in which the nodes potentially have the most elements. This analysis helps in estimating the actual memory requirement in an application.

9.5.2 Problems suitable for solution synthesis

All solution synthesis techniques described in this chapter construct the set of all solutions. Therefore, their usefulness is normally limited to CSPs in which all the solutions are required.

The amount of computation involved in solution synthesis is mainly determined by the sizes of the nodes. In general, the looser a CSP is, the more compound labels are legal, and consequently more computation is required. The tighter a problem is, the fewer compound labels there are in each node, and consequently less computation is required. This suggests that solution synthesis methods are more useful for tightly constrained problems.

In Chapter 2, we classified CSP solving techniques into problem reduction, search-

ing and solution synthesis. Now we have looked at all three classes of techniques, we shall study their applicability in the classes of problems whown in Table 2.1 of Chapter 2.

If a single solution is required, then a loosely constrained CSP can easily be solved by any brute force search: relatively many solutions exist in the search space, and therefore few backtracking can be expected. However, when the problem is tightly constrained, naive search methods such as Chronological Backtracking may require a large number of backtracks. In such problems, problem reduction methods could be useful. Besides, since the problem is tightly constrained, efforts spent in propagating the constraints are likely to result in successfully reducing the domains and constraint sizes.

With search methods, finding all solutions basically requires one to explore all parts of the search space in which one cannot prove the non-existence of solutions. As in CSPs which require single solutions, the tighter the problem, the more effective problem reduction methods are in pruning off the search space. Besides, as explained above, the tighter the CSP, the fewer elements one could expect to be included in the nodes constructed by both Freuder's and Essex solution synthesis algorithms, and therefore the more efficient these algorithms could be expected.

When the search space is large and the problem is loosely constrained, finding all solutions is hard. Both problem reduction and solution synthesis methods cannot be expected to perform much better than brute force search in this class of CSPs. Table 9.1 summarizes our analysis in this section.

Table 9.1 Mapping of tools to problems

Solutions required	Tightness of the problem	
	Loosely constrained	Tightly constrained
Single solution required	Problem is easy by nature; brute force search (e.g. simple backtracking) would be sufficient	Problem reduction helps to prune off search space, hence could be used to improve search efficiency
All solutions required	When the search space is large, the problem is hard by nature	Problem reduction helps to prune off search space; solution synthesis has greater potential in these problems than in loosely constrained problems

9.5.3 Exploitation of advanced hardware

Part of the motivation for developing AB is to exploit the advances in hardware development. Although solution synthesis is memory demanding by nature, this problem has been alleviated by the fact that computer memory has been made much cheaper and more abundant in recent years. Besides, the wider availability of cheaper content-addressable memory and parallel architectures make solution synthesis a more probable tool for CSP solving than, say, ten years ago. In this section, we shall explain how AB can be helped by these advanced hardware developments. Although the use of advanced hardware does not change the complexity of AB, it does affect the real computation time.

In AB, each node of order k where $k > 1$ is constructed by two nodes of order $k - 1$. As soon as these two nodes have been constructed, the node of order k can be constructed. Therefore, there is plenty of scope for parallelism in the construction of nodes.

The efficiency of the Compose procedure could be improved with the help of content-addressable memory. Let us assume that P and Q are nodes for the variables $\{x, x_1, ..., x_m\}$ and $\{x_1, ..., x_m, y\}$, respectively. When P and Q are used to construct the node R (which is a node for the variables $\{x, x_1, ..., x_m, y\}$), the following operation is involved: given any tuple ($<x,a><x_1,v_1>...<x_m,v_m>$) in P, one needs to retrieve all tuples of the form ($<x_1,v_1>...<x_m,v_m><y,b>$) from Q before one can check whether $<x,a>$ and $<y,b>$ are compatible. This retrieval involves going through all the tuples in Q and performing pattern matching on each of them. With content-addressable memory, one needs no indexing, and therefore can retrieve the tuples directly.

One system which partially meets the requirements of the Essex Algorithms is the Intelligent File Store (IFS). It provides content-addressable memory and parallel search engines, and therefore is capable of returning all the tuples which match the required pattern in roughly constant time. Unfortunately, it does not facilitate parallel construction of the nodes.

9.6 Concluding Remarks

Solution synthesis involves constructively building up compound labels for larger and larger groups of variables. Solution synthesis in general is more useful for tightly constrained problems in which all solutions are required.

In this chapter, three solution synthesis algorithms have been explained: Freuder's algorithm, the invasion algorithm, and the Essex Algorithms (AB and its variants). Freuder's solution synthesis algorithm is applicable to CSPs with general constraints. The basic idea is to incrementally construct a lattice, which we call the

minimal problem graph, or MP-graph, in which every node contains the set of *all* legal tuples for a unique subset of variables. The node for a set of k variables S is constructed using the k-constraint on S (if any) and all the nodes for the subsets of $k-1$ elements of S.

The invasion algorithm is applicable to binary constraint CSPs, though it can be extended to handle general constraints with additional complexity. It exploits the topology of the constraint graph, and is especially useful for problems in which each variable is involved in only a few constraints. Starting with the 0-compound label, the basic principle is to extend each compound label of the last iteration by adding to it a label for a new variable. In the process of doing so, a solution graph is created to store all the solutions. We have pointed out the close relationship between the invasion algorithm and the minimal bandwidth ordering (MBO).

The Essex Algorithms are also more suitable for binary constraint CSPs, but can be extended to handling general constraints. The idea is to reduce both the number of nodes and the complexity of nodes construction in Freuder's algorithm. This is done by ordering the variables, and constructing nodes only out of adjacent nodes. It is argued that the efficiency of the Essex Algorithms can be significantly improved by employing a parallel machine architecture with content-addressable memory.

9.7 Bibliographical Remarks

The idea of solution synthesis was first introduced by Freuder [1978]. The invasion algorithm was proposed by Seidel [1981]. In this chapter, we have pointed out the relationship between Seidel's work and the minimal bandwidth ordering (MBO), described in Chapter 6. Algorithms which take a polynomial time to find the minimal bandwidth were first published by Saxe [1980], and then improved by Gurari & Sudborough [1984]. The Essex Algorithms are reported by Tsang & Foster [1990]. The IFS was developed by Lavington *et al*. [1987, 1988, 1989].