Chapter 7
Exploitation of problem-specific features

7.1 Introduction

In Chapter 2, we explained that if there are n variables in a CSP, and the maximum size of the domains of the variables is a, one would have to deal with a search tree with $O(a^n)$ leaves. Therefore, worst case time complexity of CSP solvers is $O(a^n)$ in general. In this chapter, we shall look at techniques which exploit the specific features of the individual problems and hopefully reduce the time complexity to below $O(a^n)$.

Not every variable is constrained by every other variable in every CSP. The topology of the constraint hypergraph or primal graph could be exploited in solving some CSPs. Most of the techniques discussed in this chapter exploit the topology of such graphs, and some of them use problem reduction techniques to reduce the complexity of the problem.

Section 7.2 discusses the possibility of decomposing problems into independent subproblems (which allows one to apply the divide and conquer strategy). Section 7.3 identifies a set of "easy problems", namely those in which constraint graphs form trees and k-trees (Definition 3-26), for which efficient algorithms exist. Section 7.4 discusses techniques to remove redundant constraints (Definitions 3-16, 3-19) to transform CSPs to equivalent but "easy" problems. Section 7.5 introduces the cycle-cutset method, which is basically a dynamic search method that switches to a backtrack-free search when the remaining problem is easy. Section 7.6 introduces the tree-clustering method, which groups the variables into clusters to form subproblems and solves the problem by solving these smaller and easier subproblems separately. Section 7.7 extends the relationship between the width of a constraint graph and k-consistency concluded in Theorem 6.1. Section 7.8 introduce specialized algorithms for handling CSPs with numerical variables and conjunctive binary constraints.

7.2 Problem Decomposition

If the variables in a CSP can be separated into independent groups (i.e. no variable in one group constrains any variable in any other group), then these groups of variables can be labelled separately. When this is the case, a smaller space needs to be searched. Figure 7.1 shows the search spaces when the problem can and cannot be decomposed. If a CSP with n variables can be decomposed into three subproblems with p, q and r variables, respectively, then the size of the search space is $O(a^p + a^q + a^r)$ rather than $O(a^{p+q+r})$, where a is the size of each domain and $p + q + r$ equals to the total number of variables in the problem n.

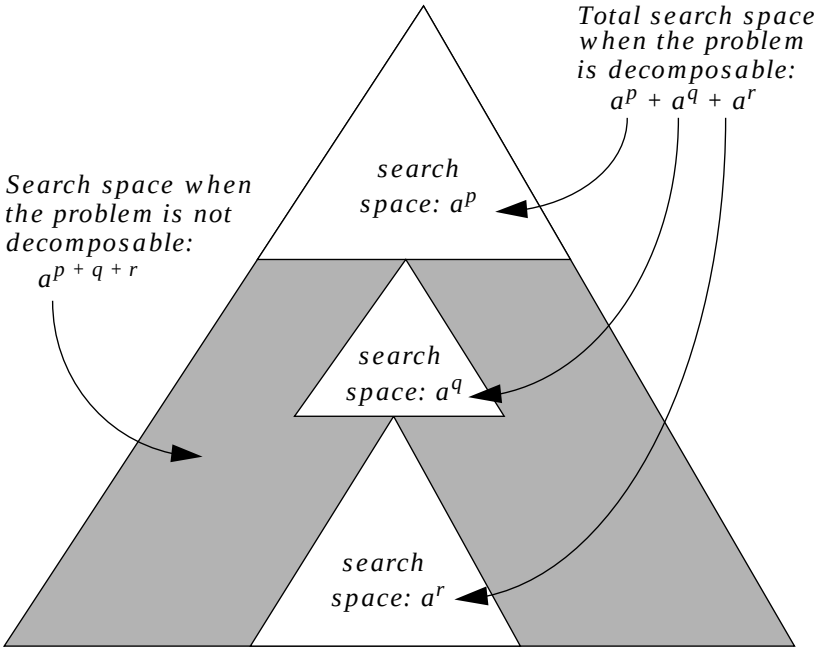


Figure 7.1 The size of the search space when a problem is decomposable. a = size of the domains of the variables and the total number of variables in the problem = $p + q + r$

A problem can be decomposed if its primal graph (Definition 4-1) is not connected (Definition 1-21). A graph with *n* nodes can be partitioned in *O(n)* time, using the Partition procedure below. Therefore, running a graph partitioning algorithm before searching for solutions does not increase the overall complexity of the search algorithm.

```
PROCEDURE Partition(V, E)
/* given a graph, partition the nodes into unconnected clusters */
BEGIN
    SS ← { };    /* SS is the set of clusters of variables to be returned */
    WHILE (V ≠ { }) DO
        BEGIN
            z ← any node in V; S ← {z}; V ← V – S; Cluster ← { };
            /* S and Cluster are used as working storage */
            WHILE (S ≠ { }) DO
                BEGIN
                    x ← any element in S; S ← S – {x};
                    Cluster ← Cluster + {x};
                    FOR each y such that (x,y) is in E AND y is in V DO
                        BEGIN
                            V ← V – {y};
                            S ← S + {y};
                            E ← E – {(x,y)};
                        END
                END /* of inner WHILE loop */
            SS ← SS + Cluster;              /* one cluster found */
        END; /* of outer WHILE loop */
    return(SS);
END /* of Partition */
```

One node is deleted from *V* in each iteration of the FOR loop, so the FOR loop can only iterate *n* times, where *n* is the number of nodes in the graph. Therefore, the time complexity of the Partition procedure is *O(n)*. Program 7.1, *partition.plg*, shows a Prolog implementation of the Partition algorithm. It assumes the graph to be stored in the Prolog database in exactly the same format as in the previous programs.

7.3 Recognition and Searching in *k*-trees

7.3.1 "Easy problems": CSPs which constraint graphs are trees

This section illustrates the fact that when the constraint graph (Definition 4-1) of a CSP is a tree, one can solve this problem in $O(na^2)$, where n is the number of variables and a is the maximum domain size in the problem. This motivates the recognition of problems which constraint graphs are trees, or reducing CSPs to problems of such class.

We mentioned in Chapter 6 that a tree is a graph with a width equal to 1. According to Theorem 3.1, if the constraint graph of a CSP is a tree, then a search for solutions in this problem is backtrack-free if node- and arc-consistency (i.e. strong 2-consistency) are maintained in it. We mentioned in Chapter 3 that, in fact, strong 2-consistency is stronger than necessary to guarantee a search to be backtrack-free in a CSP which constraint graph is a tree. All one needs to achieve is DAC in the CSP.

We shall prove that the Tree_search procedure below can be used to solve CSPs which constraint graphs are trees in $O(na^2)$:

```
PROCEDURE Tree_search((Z, D, C))
/* The constraint graph of (Z, D, C), G((Z, D, C)), is a tree */
BEGIN
    Give the variables an ordering < such that all parents are placed
        before their children in G((Z, D, C));
    achieve NC and DAC in (Z, D, C, <);
    Labelled ← { };
    /* backtrack-free search */
    WHILE Z ≠ { } DO
        BEGIN
            x ← the frontmost variable in Z according to <;
            Z ← Z – {x};
            v ← a value in D_x such that <x,v> is compatible with all the
                labels in Labelled;
            Labelled ← Labelled + {<x,v>};
        END;
    return( Labelled );
END /* of Tree_search */
```

Theorem 7.1 (due to Dechter & Pearl, 1988a)

Given a CSP P, if the constraint graph of P forms a tree, then P can be solved in $O(na^2)$.

Proof

Given a CSP (Z, D, C), if its constraint graph forms a tree, then we can order the variables in such a way that all the parents are placed before all their children (this ordering can be obtained by a preorder search in the tree). Let this ordering be $<$. We can show that after maintaining NC and DAC in (Z, D, C, $<$), then we can label all the variables without backtracking. If some domain are reduced to empty sets after problem reduction, then no search is needed and failure can be reported. Otherwise, the problem is 1-satisfiability by definition. Given that the constraint graph forms a tree, every variable x is constrained by at most one other variable y such that $y < x$. Given the ordering $<$, y would have already been labelled when x is being labelled. Since the reduced problem is 1-satisfiable, NC and DAC, we can always find a value for x which is compatible with y's label. That means we can label every variable without needing to revise its parent's label.

The complexity of a backtrack-free search is $O(na)$, where n is the number of variables and a is their maximum domain size. This is because in the worst case, all one needs to do is to go through all the values of each variable to find a value which is compatible to all the labelled variables.

By using the NC-1 and DAC-1 procedures in Chapter 4, NC and DAC in a tree-type constraint graph can be achieved in $O(na)$ and $O(na^2)$ time, respectively. Therefore, the time complexity of solving a CSP which constraint graph forms a tree is dominated by complexity of the DAC achievement procedure, i.e. $O(na^2)$.

(Q.E.D.)

The procedure Acyclic recognizes acyclic undirected graphs (i.e. trees) in *O(n)*, where n is the number of variables in the problem.

```
PROCEDURE Acyclic(V, E)
/* Return True if the graph (V, E) is acyclic, return False otherwise */
BEGIN
   WHILE (V ≠ { }) DO
      BEGIN
         y ← any node in V;
         S ← {y};
         WHILE (S ≠ { }) DO
            BEGIN
               z ← any node in S; S ← S – {z};
               V ← V – {z};
               FOR each x adjacent to z with regard to E DO
                                    /* (x,z) ∈ E */
```

```
                    BEGIN
                        IF (x is in S) THEN return(False);
                        E ← E – {(x, z)};
                        /* note that (x, z) is the same object as (z, x) */
                        S ← S + {x};
                    END
                END
            END
        return(True);
    END /* of Acyclic */
```

The Acyclic procedure starts from an arbitrary node *y* in the graph (*V, E*). *S* is the set of all nodes which are adjacent to *y*. In every iteration of the inner WHILE loop, the Acyclic procedure removes one node from *S*, together with all the edges joining it. Besides, it checks whether this node is adjacent to any other node in *S*. If it is, then a cycle is found, and the Acyclic procedure will report failure. The outer WHILE loop handles one cluster in each iteration in case the graph is not connected. For a graph with *n* nodes, there will be exactly *n* iterations in the inner loop when the graph is acyclic. When the graph is cyclic, fewer iterations may be needed. Therefore, the time complexity of Acyclic is *O(n)*. Program 7.2, *acyclic.plg*, shows a Prolog implementation of this algorithm.

7.3.2 Searching in problems which constraint graphs are *k*-trees

In Chapter 3, we introduced the concept of *k*-trees (Definition 3-26), which is a generalization of trees. Let *n* be number of variables in the problem and *a* be the maximum size of the domains. Freuder points out that if the constraint graph of a problem can be recognized as a *k*-tree, then it can be solved in $O(na^{k+1})$ time. This can be achieved by first finding an ordering which induced-width (Definition 4-5) is *k*, and then achieving adaptive-consistency in the problem.

7.3.2.1 Recognition of *k*-trees

Let us first introduce a procedure W which, given a graph and an integer *k*, determines whether the graph is a *k*-tree, and returns an ordering of the nodes such that the induced-width of the graph equals *k*.

```
PROCEDURE W( (V, E), k );
/* Given a constraint graph G = (V, E) of a constraint satisfaction
    problem P and an integer k, return an ordering < of V such that
    induced-width(P, <) = k if G is a k-tree; NIL if it is not */
BEGIN
    /* initialization */
```

```
K ← { }; Sum ← 0;
FOR each node x in E DO
    BEGIN
        Count[x] ← the degree of x;
        IF (Count[x] = k) THEN K ← K + {x};
        Sum ← Sum + Count[x];
    END
IF (Sum ≠ 2nk – k – k^2) THEN return(NIL);    /* note[1] */
/* major computation */
FOR i = n to k + 1 by –1 DO
    IF (K = { }) THEN return(NIL);
    ELSE BEGIN
                v ← any node in K; K ← K – {v};
                V' ← neighbourhood(v);
                E' ← {(a,b) | (a,b) ∈ E ∧ a, b ∈ V'};
                IF ((V', E') is a complete graph) THEN
                    BEGIN
                        Ordering[i] ← v; V ← V – {v};
                        FOR each w such that (v, w) ∈ E DO
                            BEGIN
                                E = E – {(v, w)};
                                Count[w] ← Count[w] – 1;
                                IF (Count[w] = k) THEN K ← K + {w};
                            END;
                    END
                ELSE return(NIL);
            END /* of ELSE */
/* at this point, all but k nodes have been ordered */
IF (the remaining k nodes form a complete graph)
THEN BEGIN
        Ordering[1] to Ordering[k] ← remaining nodes in any order;
        return(Ordering);
    END
ELSE return(NIL);
END /* of W */
```

The procedure W is in fact providing a constructive proof to the proposition that the input graph is a *k*-tree. According to the definition, a *k*-tree should have exactly $k(k-1)/2 + (n-k)k$ edges, where n is the number of nodes in the *k*-tree. This is because a *k*-tree must contain a complete graph of k nodes, which has $k(k-1)/2$ edges. Besides the nodes in this complete graph, there should be $(n-k)$ other nodes,

1. A typographical error in [Freu90,p.6] has been corrected here.

each of them having exactly k edges (according to the definition), making $(n - k)k$ edges in total. So the total number or edges in a k-tree is $k(k - 1) / 2 + (n - k)k$. In the W procedure, each edge is counted twice (once from each end). That is why after the degree of each node is counted, the *Sum* is checked against $2 \times (k(k - 1) / 2 + (n - k)k) = 2nk - k - k^2$.

It should be noted that the ordering of the nodes in the set K is unimportant. This is because if the graph is indeed a k-tree, then no two nodes in K should be adjacent to each other (otherwise the removal of one of them would cause the degree of the other to be reduced below k).

The initialization goes through the nodes once, and therefore takes $O(n)$ time to compute. The second part of the procedure removes one node in each iteration. Therefore, if we assume that both counting the number of edges and testing whether a graph is complete (given the degree of each node) takes a constant time, then the main FOR loop takes $O(n)$ time to compute. So the whole procedure W takes $O(n)$ time to compute.

Figure 7.2 shows an example of the W procedure in action. The input graph (taken from Figure 3.6(d)) is recognized as a 3-tree, and an ordering is found which induced-width is equal to 3.

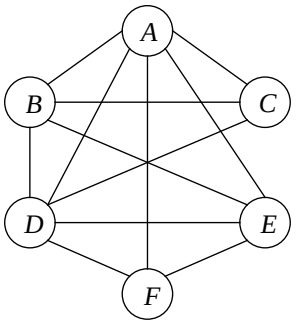
Theorem 7.2

A k-tree constraint graph with n nodes can be recognized as a k-tree, and an ordered constraint graph with induced-width k can be found (or k-1 for trivial k-trees), in O(n) time.

Proof

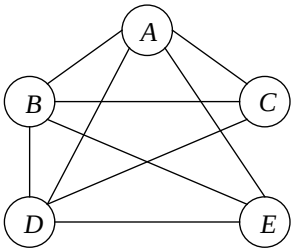
Procedure W will recognize k-trees and return an ordering in $O(n)$, where n is the number of nodes in the input graph. If the graph is a trivial k-tree (i.e. it has k nodes and is complete), then any ordering of the nodes would have induced-width equal to $k - 1$. If the constraint graph of a CSP is a nontrivial k-tree, then the induced-width of this CSP under the ordering returned by W is k for the following reasons. Procedure W ensures that for every $j > k$, the j-th node has exactly k nodes before it. Moreover, these k nodes form a complete graph, and therefore, no edge needs to be added between them when adaptive-consistency is maintained. Therefore, the induced-width of the graph under the ordering returned by procedure W is k.

(Q.E.D.)



$K = \{C, F\}$

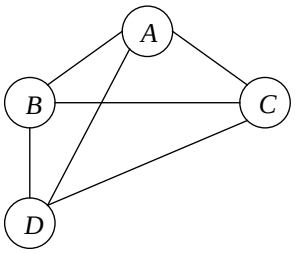
(a) An input graph to be recognized as a 3-tree (value of K after initialization of procedure W)



ordering: F

$K = \{C, E\}$

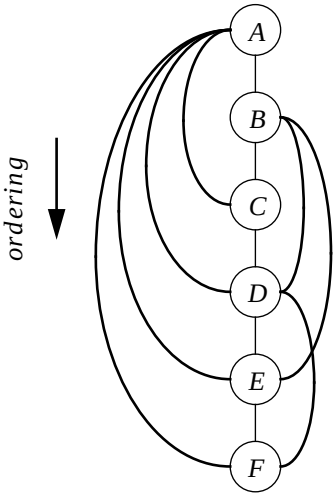
(b) The graph, ordering and K after the first step of procedure W



ordering: E → F

$K = \{A, B, C, D\}$

(c) The graph, ordering and K after the second step of procedure W (any node can be chosen in the next step, which is skipped)



(d) Ordering produced by W (achieving adaptive-consistency will not add extra edges to it)

Figure 7.2 Steps for recognizing a 3-tree and ordering the nodes

7.3.2.2 Solving CSPs which constraint graphs are k-trees

Theorem 7.3

Let G be the primal graph of a CSP P. If G is a trivial k-tree, then the induced-width of P is $k-1$. If G is a non-trivial k-tree, then the induced-width of P is k. Whenever G is a k-tree, the induced-width of P is equal to G's width:

$\forall$ csp(P): trivial_k-tree(G(P)) $\Rightarrow$ induced-width(P) = $k-1$
$\forall$ csp(P): k-tree(G(P)) $\Rightarrow$ induced-width(P) = k
$\forall$ csp(P): induced-width(G(P)) = width(G(P))

Proof

By definition, the induced-width of a CSP P is at least as great as the width of G(P) under any ordering. If the primal graph of P is a trivial k-tree, then its induced-width is $k-1$ because under any ordering of the nodes, the final node will be adjacent to $k-1$ nodes before it. If the primal graph of P is a non-trivial k-tree T, then there exists a node which neighbourhood is a complete graph of k nodes. So the width of T, hence the induced-width of T, is no less than k. On the other hand, the induced-width of the ordering produced by algorithm W is k, so the width of T cannot be greater than k. Since the width of T is no less than and no greater than the induced-width of T, the width must be equal to the induced-width of T.

(Q.E.D.)

Theorem 7.3 implies that achieving adaptive-consistency on a k-tree structured CSP under the ordering returned by procedure W does not change the width of the constraint graph. This can be seen from a different perspective: since every node in the ordering return by procedure W is adjacent to exactly k preceding nodes which form a complete graph, achieving adaptive-consistency according to this ordering does not increase the number of edges in the constraint graph. Consequently, the width of the reduced problem will not be changed.

The k-tree_search procedure below shows one way of exploiting the fact that the CSP's constraint graph is a k-trees:

```
PROCEDURE k-tree_search(Z, D, C)
/* given a CSP, finds a solution to it in O(n^(ak+1)) time if its constraint
   graph is a k-tree for some k; otherwise, return NIL */
BEGIN
```

```
    /* check if the constraint graph is a k-tree for any k */
    k ← 1;
    REPEAT
        Ordering ← W((Z, D, C)), k);
        k ← k + 1;
    UNTIL (Ordering ≠ NIL) OR (k > |Z|);
    /* one might want to further limit the value of k above */
    IF (Ordering = NIL)
    THEN return(NIL)     /* other methods needed to solve the CSP */
    ELSE BEGIN
                P ← Adaptive_consistency(Z, D, C, Ordering);
                Result ← perform backtrack-free search on P;
                return(Result);
            END
END /* of k-tree_search */
```

Basically, the procedure *k*-tree_search detects whether a *k* exists such that the CSP's primal graph is a *k*-tree. Then it achieves adaptive-consistency in the input problem before performing a backtrack-free search.

Theorem 7.4

A CSP which constraint graph is a k-tree can be solved in $O(na^{k+1})$ time and $O(na^k)$ space, where n is the number of variables in the problem, and a is the maximum domain size in the problem.

Proof

The *k*-tree_search procedure would prove the point. Given a CSP, if its constraint graph is a *k*-tree, then it takes $O(n^2)$ time to recognize it. This is because in the worst case, one has to go through all the *n* values to find *k*, and procedure W takes $O(n)$ to compute.

Achieving adaptive-consistency requires $O(na^{W^*+1})$, where W^* is the induced-width of the constraint graph. When the graph is a non-trivial *k*-tree, its induced-width is *k*; so achieving adaptive-consistency requires $O(na^{k+1})$.

A backtrack-free search takes $O(na)$ time to complete. Combining all three steps, the time complexity of procedure *k*-tree_search is $O(na^{k+1})$.

The space required by Adaptive_consistency is $O(na^k)$, which dominates the space complexity of *k*-tree_search.

(Q.E.D.)

According to our definitions in Chapter 3, any graph is a partial *k*-tree for a sufficiently large *k*. Besides, a partial *k*-tree can be transformed into a *k*-tree by adding to it a sufficient number of redundant constraints. Therefore, it appears that one can use the *k*-tree_search procedure to tackle general CSPs. However, there are no efficient algorithms for recognizing partial *k*-trees for general *k*.

7.4 Problem Reduction by Removing Redundant Constraints

In Chapter 3, we introduced the concept of redundant constraints. It is possible to reduce a CSP to an "easy problem" (as defined in the last section) by removing redundant constraints. In Chapter 3, we pointed out that identifying redundant constraints is hard, in general. However, as in the case of removing redundant labels and redundant compound labels, some redundant constraints may be easier to identify than others. For example, a constraint in a binary CSP can be removed if it is *path-redundant* (Definition 3-19) — if S is the set of all *path-induced* (Definition 3-18) compound labels for x and y, then the constraint $C_{x,y}$ is redundant if S is a subset of $C_{x,y}$. The procedure Path_redundant below detects the path-redundancy of any given binary constraint:

```
PROCEDURE Path_redundant((x ,y), P)
/* P is a CSP and x, y are two variables in it */
BEGIN
    (Z, D, C) ← NC-1(P);          /* achieve node-consistency in P */
    U ← {(<x,a><y,b>) | a ∈ D_x ∧ b ∈ D_y};
    FOR each z in Z such that x ≠ z AND y ≠ z DO
        BEGIN
            DisAllowed_CLs ← U – C_x,y
            For each cl ∈ DisAllowed_CLs DO
                IF NOT Permitted(cl, z, D_z, C_x,z, C_z,y)
                THEN C_x,y ← C_x,y + {cl};
            IF C_x,y = U THEN return(True)
        END
    return(False);
END /* of Path_redundant */


PROCEDURE Permitted((<x,a><y,b>), z, D_z, C_x,z, C_z,y)
BEGIN
    FOR each c ∈ D_z DO
        IF (satisfies((<x,a><z,c>), C_x,z) AND
                satisfies((<z,c><y,b>), C_z,y))
```

```
        THEN return(True);
    return(False);
END /* of Permitted */
```

Path_redundant checks whether every compound label which is disallowed by $C_{x,y}$ is indeed disallowed by at least one path. Only if this is the case could $C_{x,y}$ be deleted without relaxing the constraints in the problem. If a compound label is found to be not Permitted by any path, then it need not be considered for another path; hence it is added to $C_{x,y}$ in the For loop of Path_redundant. If all the compound labels are added into $C_{x,y}$, then it is proved that the set of path-induced compound labels is a subset of $C_{x,y}$, which means the input $C_{x,y}$ is redundant ($C_{x,y}$ is a local variable in Path_redundant; its change of value within this procedure is not supposed to affect the calling program).

Let n be the number of variables in the CSP, and a the maximum domain size in the problem. The time complexity of Permitted is $O(a)$ as it examines all the a values in D_z. Path_redundant examines all pairs of values for x and y, and in the worst case, checks every compound label with every variable z. Therefore, the time complexity of Path_redundant is $O(na^3)$.

No one suggests that all the constraints in the problem should be examined. One should only run the Path_redundant procedure on those constraints which, once removed, could result in the problem being reduced to an easier problem. As mentioned before, one may attempt to remove certain constraints in order to reduce to the problem to one which is decomposable into independent problems, or to one which constraint graph is a tree. However, since not every problem can be reduced to decomposable problems or problems with their constraint graph being trees, one should judge the likelihood of succeeding in reducing the problem in order to justify calling procedures such as Path_redundant. Domain knowledge may be useful in making such judgements.

7.5 Cycle-cutsets, Stable Sets and Pseudo_Tree_Search

7.5.1 The cycle-cutset method

The *cycle-cutset method* is basically a dynamic search method which can be applied to binary CSPs. (Although extending this method to general CSPs is possible, one may not benefit very much from doing so.) The goal is to reduce the time complexity of solving the problem to below $O(a^n)$, where n is the number of variables and a is the maximum domain size in the problem. The basic idea is to identify a subset of variables in the problem where removal will render the constraint graph being acyclic. In general, the cycle-cutset method is useful for problems where most variables are constrained by only a few other variables.

Definition 7-1:

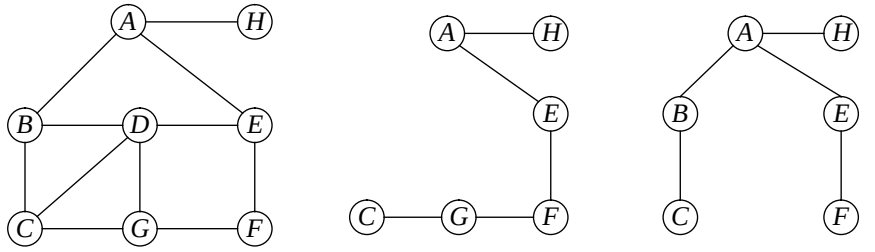
The **cycle-cutset** of a graph is a subset of the nodes in the graph which once removed, renders the graph as acyclic:

$\forall$ graph((V, E)): $\forall\ S \subseteq V$: cycle-cutset(S, (V, E)) $\equiv$ acyclic($(V - S, E')$)

where $E' = \{(a, b) \mid (a, b) \in \mathrm{E} \wedge a, b \in V - S)\}$ ■

The cycle-cutset method partitions the variables into two sets, one of which is a cycle-cutset of the CSP's constraint graph. In the case when the constraint graph is connected, the removal of the cycle-cutset renders the constraint graph to be a tree. (If the graph is not connected, the problem can be decomposed, as explained in Section 7.2). In the graph in Figure 7.3(a), two examples of cycle-cutset are {*B*, *G*} and {*D*, *G*}. The graphs after the removal of these cutsets are shown in Figures 7.3(b) and (c).

There is no known efficient algorithm for finding the minimum cycle-cutset. One heuristic to find such sets is to use the reverse of a minimum width ordering or a maximum cardinality ordering (Section 6.2.1, Chapter 6). A less laborious way is to order the variables by their degrees in descending order instead of using the minimum width ordering. The CCS procedure (CCS stands for Cycle Cut-Set) shows one possible way of using the cycle-cutset concept to solve binary CSPs.



(a) Example of a graph

(b) The tree after the removal of the cut-set {*B*, *D*}

(c) The tree after the removal of the cutset {*D*, *G*}

Figure 7.3 Examples of cycle-cutset

```
PROCEDURE CCS(Z, D, C)
BEGIN       /* Ordering is an array 1..|Z| of nodes */
   /* preprocessing — identify a cycle-cutset */
   Ordering ← elements in Z ordered by descending order of their
      degrees in the constraint graph;
   Graph ← constraint graph of (Z, D, C);
   Cutset ← { }; i ← 1;
   WHILE (the graph of (Z, D, C) is cyclic) DO
      BEGIN
         Cutset ← Cutset + {Ordering[i]};
         remove Ordering[i] and edges involving it from Graph;
         i ← i + 1;
      END

   /* labelling */
   CL1 ← compound label for variables in Cutset satisfying all con-
      straints;
   REPEAT
      FOR j ← i to |Z| DO
         remove from D_Ordering[j] values which are incompatible with
            CL1;
      Achieve DAC in the remaining problem;
      IF (the remaining problem is 1-satisfiable) THEN
         BEGIN
            CL2 ← label the remaining variables using a backtrack-
               free search;
            return(CL1 + CL2);
         END
      ELSE CL1 ← alternative consistent compound label for the
         Cutset, if any;
   UNTIL (there is no alternative consistent compound label for the
      Cutset);
   return(NIL);
END /* of CCS */
```

The cycle-cutset method does not specify how the problem in the cutset should be solved — this is reflected in the CCS procedure. Besides, orderings other than the one used in the CCS procedure can be used.

The acyclicity of a graph with n nodes can be determined in $O(n)$ using the Acyclic procedure (Section 7.3.1). The WHILE loop in the preprocessing part of CCS will iterate n times because it removes one node from Z per iteration. In each iteration, it checks whether the graph is acyclic. Therefore, the preprocessing part of the algorithm takes $O(n^2)$ time to complete. In the worst case, the time complexity of find-

ing a consistent compound label CL1 is $O(a^c)$ time, where a is the size of each domain, and c is the number of variables in the Cutset. CL2 is found using the Tree_search procedure. Therefore, the time complexity of finding CL2 is $O((n - c)a^2)$, or $O(na^2)$ in the worst case. The overall complexity of the procedure CCS is therefore $O(na^{c+2})$.

Figure 7.4 shows the procedure for applying the cycle-cutset method to an example CSP. The cycle-cutset method can be used together with search strategies which use dynamic ordering. However, using the cycle-cutset method with such search strategies incurs the overhead of checking, after labelling every variable, whether the constraint graph of the unlabelled variables is acyclic.

The cycle-cutset method can be extended to handle general constraints. This can be done by generating the primal graph before identifying a cycle-cutset. However, this would mean that whenever a k-ary constraint C is present, the cutset must contain at least $k - 2$ of the variables involved in C.

When the cycle-cutset method is used together with chronological backtracking, a smaller space will normally be searched, due to the use of DAC after the cycle cutset is labelled (see Figure 7.5). This may also be true when this method is used together with other search strategies. However, in some search strategies, for example, those which learn and those which look ahead and order the variables dynamically, the search sequence may be affected by the history of the search. It is thus not guaranteed that the cycle-cutset method will explore a smaller search space when coupled with these methods.

Besides, in order to minimize the complexity of the CCS procedure, it may be tempting to use the cutset with the minimum size (if one can find it). However, variables in the minimum cutset tend to be unconnected. That means less constraint propagation is possible when labelling the variables in the cutset. Furthermore, there are potentially more consistent compound-labels for the cycle-cutset than for an average set of variables of the same size in the problem. Therefore, placing this cycle-cutset at the front of the ordering of the variables, no matter how the variables in the cutset are ordered, may not always benefit a search more than using the heuristics described in Chapter 6 (such as the minimum width ordering, minimum bandwidth ordering and the Fail First Principle).

Whether the cycle-cutset method is effective in realistic problems has yet to be explored.[2]

---

2. The cycle-cutset method has been tested on *small* randomly generated CSPs (with maximum 15 variables and 9 values each), and is shown to out-perform BT (Section 5.2.1, Chapter 5) by 20% in terms of consistency checks [DecPea87]. Such a result does not give much support to the efficiency of this method in any realistic applications.

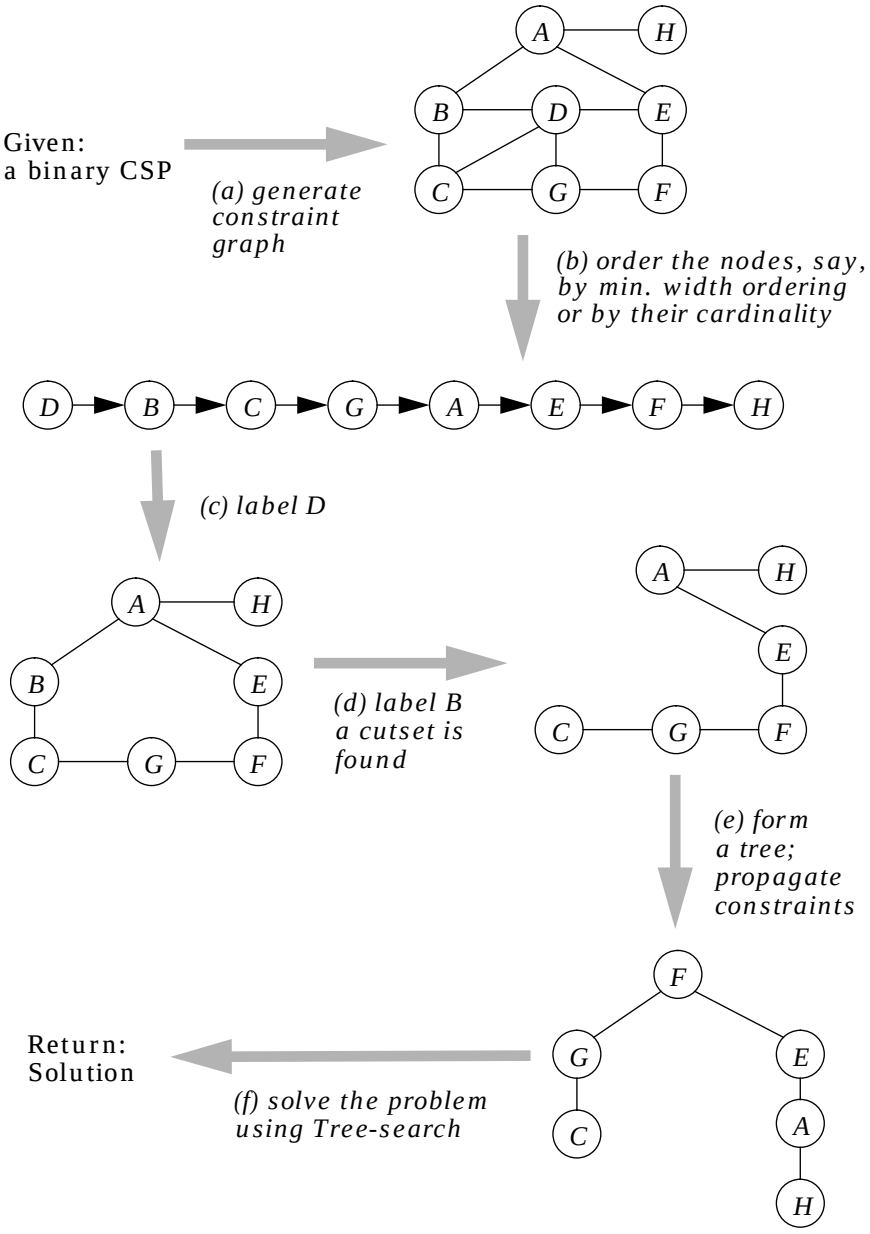


Figure 7.4 Procedure of applying the cycle-cutset method to an example CSP

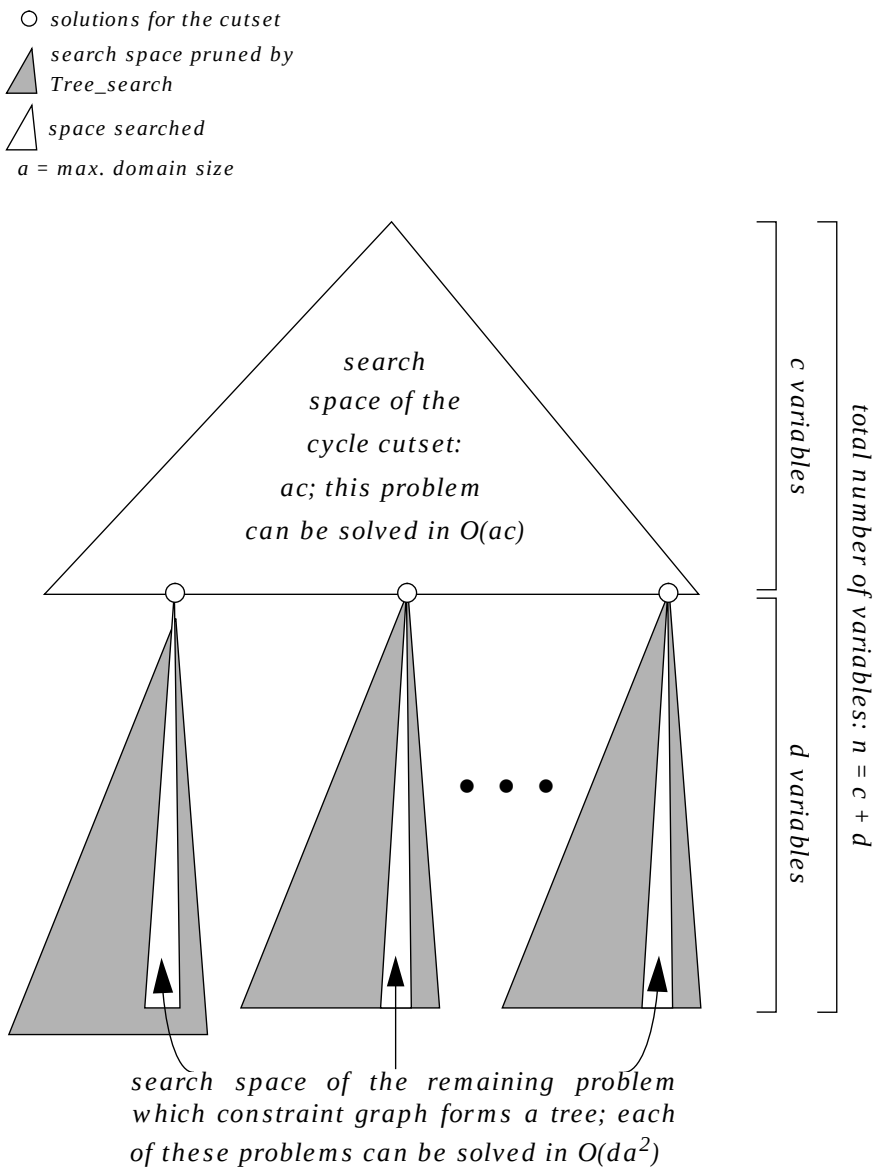


Figure 7.5 Search space of the cycle-cutset method. The overall complexity of the problem is $O(da^{c+2})$, where a is the maximum domain size, c is the size of the cycle cutset, and d is the number of variables not in the cycle cutset

7.5.2 Stable sets

One may also exploit the topology of the primal graph by identifying **stable sets**. The principle is to partition the nodes in the primal graph into sets of mutually independent variables, so that they can be tackled separately.

Definition 7-2:

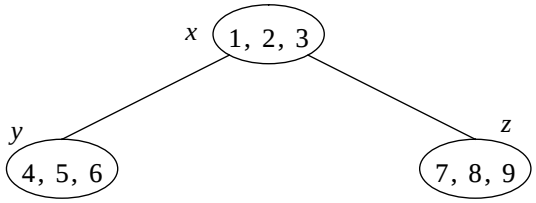
A **stable set** S of a graph G is a set of non-overlapping sets of nodes in G such that no edge joins any two elements of different sets in S:

$\forall$ graph((V, E)): $\forall\ S \subseteq \{s \mid s \subseteq V\}$:
stable_set(S, (V, E)) $\equiv$
$(\forall\ s_1, s_2 \in S: (s_1 \cap s_2 = \{\} \wedge (\forall\ x \in s_1, y \in s_2: (x, y) \notin E)))$ ■

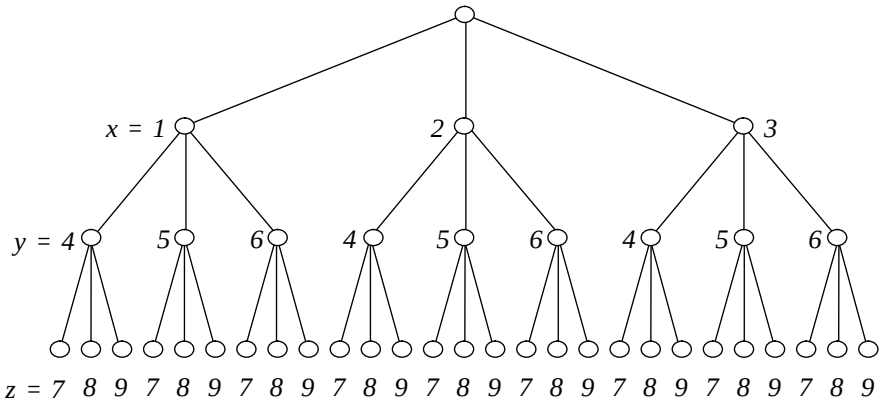
The motivation for identifying stable sets can be seen from the example given in Figure 7.6. Given variables x, y and z and their domains as shown in Figure 7.6(a), a simple backtracking search which assumes the ordering (x, y, z) have a search space with ($3 \times 3 \times 3$ =) 27 tips in the search tree, (as shown in Figure 7.6(b)). But since y and z are independent of each other, the search space could be seen as an AND/OR tree, as shown in Figure 7.6(c). On failing to find any label for any of the subtrees which is compatible with the label <x,1>, x will be backtracked to and an alternative value will be tried.

The stable set in this example comprises sets of single variables. In general, the sets in a stable set for a CSP may contain more than one variable. Assume that the variables are ordered in such a way that after labelling r variables, the rest of the variables can be partitioned into clusters that form a stable set. After a legal compound label CL for the r variables is found, the clusters can be treated as separate problems. If there exists no compound label for the variables in any of these clusters which is compatible with CL, then CL is discarded and an alternative compound label is found for the r variables. This process continues until compatible compound labels are found for the r variables and the individual clusters. A crude procedure which uses this principle is shown below:

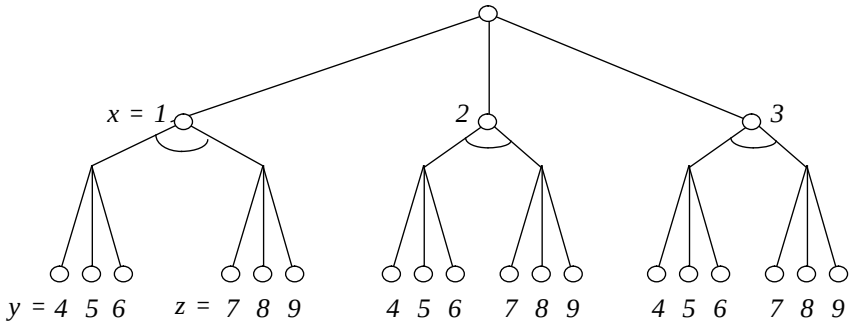
```
PROCEDURE Stable_Set(Z, D, C)
/* Given a CSP, Stable_Set returns a solution tuple if it is found; NIL
   otherwise; how the stable set SS is found is not suggested here */
CONSTANT r;
BEGIN
   /* initialization */
   Ordering ← an ordering of the variables in Z;
   R ← the set of the first r variables in the Ordering;
   SS ← stable set containing the rest of the variables partitioned;
```



(a) Example of a constraint graph



(b) Search space for the CSP in (a) under simple backtracking



(c) Search space for the CSP in (a) when stable set is considered (the search space forms an AND/OR tree)

Figure 7.6 Example illustrating the possible gain in exploiting stable sets

```
    /* labelling starts */
    CL ← legal compound label for the variables in R;
    REPEAT
        FOR each Si in SS DO
            Labeli ← compound label for Si which is compatible with
                CL;
        IF (Labeli for some i is not found)
        THEN CL ← alternative legal compound label for the variables
            in R, if any ;
        ELSE return(SS + Labeli for all i);
    UNTIL (there is no alternative legal compound label for the vari-
        ables in R);
    return(NIL); /* signifying no solution */
END /* of Stable_Set */
```

For simplicity, we assume that the complexity of ordering the variables and partitioning the graph are relatively trivial compared with the labelling part of the algorithm. The worst case time complexity for finding a legal compound label *CL* for the *r* variables is in the general $O(a^r)$, where *a* is the maximum domain size in the problem. Let the size of the *i*-th cluster be s_i, and *s* be the maximum value of all s_i. The complexity for finding a legal compound label for all clusters is in general $O(a^s)$. Therefore, the complexity of the whole problem is $O(a^{r+s})$. The space searched under the Stable_Set procedure is shown in Figure 7.7.

Unfortunately, there is no known algorithm for ordering the variables so as to minimize *r* + *s*. The algorithm Stable_Set above does not specify how the variables should be ordered. One may choose different sizes (*r*) for the set *R* in the Stable_Set procedure above. The maximum size of the stable set (*s*) may vary depending on the Ordering and the *r* chosen.

7.5.3 Pseudo-tree search

Another algorithm which uses a similar idea as the stable sets is the **pseudo-tree search algorithm**. It uses a similar principle as the graph-based backjumping algorithm described in Chapter 5. When a variable *x* cannot be given any label which is compatible with the compound label committed to so far, both pseudo-tree search and graph-based backjumping will backtrack to the most recent variable *y* which constrains *x*. The major difference between these two algorithms is that when backtracking takes place, Graph-based BackJumping will undo all the labels given to the variables between *y* and (including) *x*. Pseudo-Tree Search will only undo those labels which are constrained by *y*. The overhead for doing so is the maintenance of the dependency relationship among the variables. The pseudo code for the pseudo-tree search algorithm is shown below:

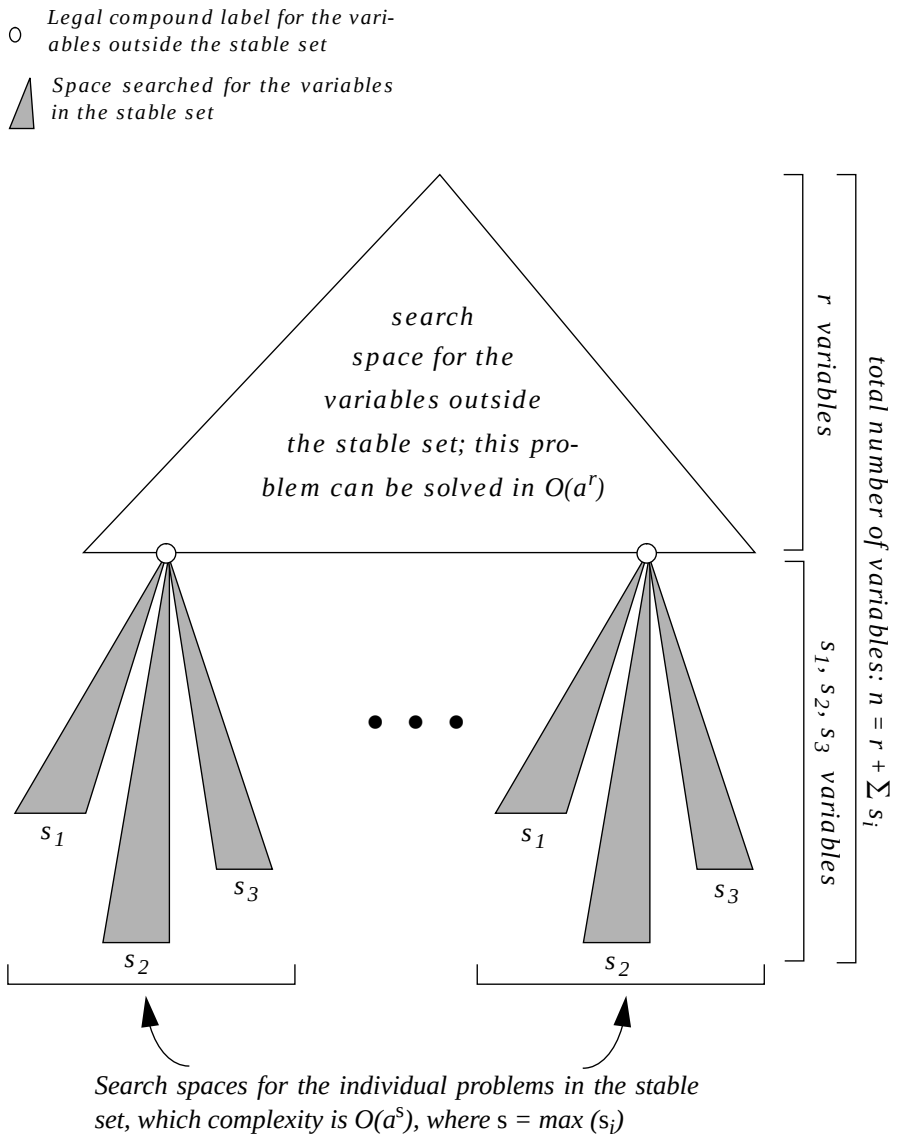


Figure 7.7 Search space of the Stable_Set procedure. The overall complexity of the problem is $O(a^{r+s})$, where a is the maximum domain size, r is the number of variables not in the stable set, and s is the size of the largest set in the stable set

```
PROCEDURE Pseudo_Tree_Search(Z, D, C);
/* for simplicity, we assume that there are n variables and all their
   domains have size m; after ordering, let Z[i] be the i-th variable,
   D[i,j] be the j-th value of variable Z[i], v[i] be an index to a value in
   the domain of Z[i] */
BEGIN
   /* initialization */
   Order the variables in Z;
   Order the values in every domain in D;
   FOR i = 1 to n DO v[i] ← 1;   /* assign first value to each variable */
   i ← 1;
   /* searching */
   WHILE (i ≤ n) DO
      /* invariance: compound label for variables Z[1] to Z[i-1] is
         legal, and v[i] is an index to a value in the domain of Z[i]
         which is yet to be examined */
      BEGIN
         IF (legal(<Z[i], D[i,v[i]]>))
         THEN i ← i + 1;
         ELSE REPEAT
                  IF (v[i] < m)
                  THEN v[i] ← v[i] + 1;
                        /* give Z[i] an alternative value */
                  ELSE BEGIN /* backtrack */
                     p ← bt_level(i);       /* to be explained in text */
                     IF (p > 0)
                     THEN FOR k = p + 1 to i DO
                              IF (Z[k] is descendent of Z[p])
                              THEN v[k] ← 1;
                              /* v[p] is to be changed */
                     ELSE return(NIL);     /* no solution */
                     i ← p;      /* backtrack to variable Z[p] */
                  END
               UNTIL (legal(<Z[i],D[i,v[i]]>) OR (i < 1));
      END; /* of WHILE */
   IF (i < 1) THEN return(NIL)
   ELSE return(values indexed by v);
END /* of Pseudo_Tree_Search */
```

For all i, $v[i]$ stores an index to the current value assigned to the variable $Z[i]$. The function legal(<$Z[i]$, $D[i,v[i]]$>) returns True if the label <$Z[i]$, $D[i,v[i]]$> is compatible with all the labels <$Z[h]$, $D[h,v[h]]$> for all $h < i$. If all the values of $Z[i]$ are incompatible with some labels committed so far, then the function *bt-level(i)* returns the greatest index j such that $Z[j]$ precedes $Z[i]$ in the Ordering, and $Z[j]$ constrains

$Z[i]$; 0 will be returned if no such $Z[j]$ exists. On the other hand, if some value for $Z[i]$ is compatible with all the labels committed to so far (i.e. $Z[i]$ has been successfully labelled but now it is backtracked to), then *bt_level*(i) returns $i - 1$. $Z[k]$ is a descendent of $Z[p]$ if (i) $C_{Z[p], Z[k]} \in C$ and $p < k$; or (ii) $Z[k]$ is a descendent of any descendent of $Z[p]$. When $Z[p]$ is backtracked to, a new value will be given to it. Therefore, all the values for $Z[k]$ have to be considered; hence the first value is given to $Z[k]$.

The Pseudo_Tree_Search procedure can be seen as a procedure which uses the stable set idea: when backtracking, variables are divided into two sets: those which require revision and those which do not.

7.6 The Tree-clustering Method

The **tree-clustering method** is useful for general CSPs in which every variable is constrained by only a few other variables. The tree-clustering method involves decomposing the problem into subproblems, solving the subproblems separately, and using the results to generate overall solutions. Interestingly, this process involves both adding and removing redundant constraints. The basic idea, which comes from database research, is to generate from the given CSP a new binary CSP whose constraint graph is a tree. Then this generated problem can be solved using the tree-searching technique introduced in Section 7.2. The solution of this generated CSP is then used to generate a solution for the original problem.

7.6.1 Generation of dual problems

Definition 7-3:

Given a problem $P = (Z, D, C)$, the **dual problem** of P, denoted by P^d, is a binary CSP (Z^d, D^d, C^d) where each variable x in Z^d represents a set (or called cluster) of variables in Z, the domain of x being the set of all compound labels for the corresponding variables in P. To be precise, for every constraint c in C, if c is a constraint on a set of variables in Z, then this set of variables in Z form a variable in P^d. There are only binary constraints in P^d, which requires the projection of the values in each cluster to the same variables in Z to be consistent:

$P^d((Z, D, C)) \equiv (Z^d, D^d, C^d)$ where:

$Z^d = \{S \mid C_S \in C\}$;

$\forall S \in Z^d: D^d{}_S =$

$\{(<x_1,v_1> \ldots <x_k,v_k>) \mid x_1, \ldots, x_k \in S \wedge v_1 \in D_{x_1} \wedge \ldots \wedge v_k \in D_{x_k}\}$;

$C^d = \{\ C^d_{S_1, S_2} \mid S_1, S_2 \in Z^d \wedge S_1 \cap S_2 \neq \{\}\ \}$, where

$C^d_{S_1, S_2} = \{(<S_1,L_1>,<S_2,L_2>) \mid$

$L_1 \in D^d_{S_1} \wedge L_2 \in D^d_{S_2} \wedge (\forall\, x \in Z\colon \forall\, v_1, v_2 \in D_x\colon$

$(\text{projection}(L_1, (<x,v_1>)) \wedge \text{projection}(L_2, (<x,v_2>))) \Rightarrow$

$v_1 = v_2)\}$ ■

The cluster for a *k*-constraint *T* thus contains the *k* variables in *T*. For example, Figure 7.8(a) shows the constraint hypergraph of a CSP:

$P = (Z, D, C)$, where:

$Z = \{A, B, C, D, E\}$
$D_A = D_B = D_C = D_D = D_E = \{1, 2\}$
$C = \{C_{A,B,C}, C_{A,B,D}, C_{C,E}, C_{D,E}\}$

P contains two 3-constraints ($C_{A,B,C}$ and $C_{A,B,D}$) and two binary constraints ($C_{C,E}$ and $C_{D,E}$), the contents of which are unimportant here. The dual problem of P is therefore:

$P^d = (Z^d, D^d, C^d)$, where:

$Z^d = \{ABC, ABD, CE, DE\}$
$D^d_{ABC} = D^d_{ABD} =$
$\{(1,1,1), (1,1,2), (1,2,1), (1,2,2), (2,1,1), (2,1,2), (2,2,1), (2,2,2)\}$
$D^d_{CE} = D^d_{DE} = \{(1,1), (1,2), (2,1), (2,2)\}$
$C^d = \{C^d_{ABC,ABD}, C^d_{ABC,CE}, C^d_{CE,DE}, C^d_{ABD,DE}\}$, where
$C^d_{ABC,ABD} \subseteq \{((1,1,1),(1,1,1)), ((1,1,1),(1,1,2)), ((1,2,1),(1,2,1)), ...\}$
$C^d_{ABC,CE} \subseteq$ .....
.....

In this example, we have used the name *ABC* to denote the newly created variable in P^d which corresponds to the variables *A*, *B* and *C* in P. The domains in P^d are compound labels in P. In this example, we have used, say, (1,1,1) as shorthand for (<*A*,1><*B*,1><*C*,1>). The constraint $C^d_{ABC,ABD}$ requires consistent values to be assigned to *A* and *B* in the original problem. Therefore, ((1,1,1), (1,1,2)) is legal as far as $C^d_{ABC,ABD}$ is concerned (because both the labels for *ABC* and *ABD* project to (<*A*,1><*B*,1>)), but ((1,1,1), (1,2,2)) is illegal (because the label for *ABC* projects to

(<*A*,1><*B*,1>) but the label for *ABD* projects to (<*A*,1><*B*,2>)). Figure 7.8(b) shows the constraint graph of the dual problem P^d. There we label the constraints with the common variables.

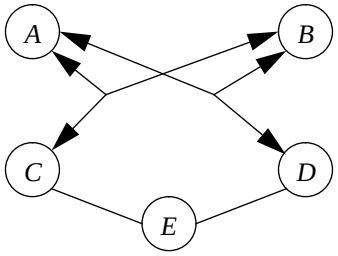
7.6.2 Addition and removal of redundant constraints

Several points are important to the development of the tree-cluster method:

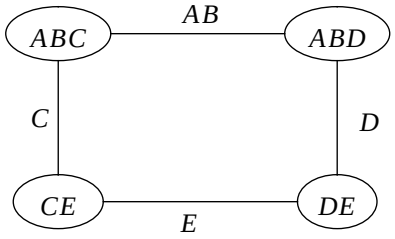
(1) Redundant constraints can be added to the original problem without changing the set of solutions; but this will change the formalization of the dual problem. For example, to the problem P shown in Figure 7.8(a), one can add a constraint $C_{C,D}$ such that $C_{C,D}$ contains all the possible compound labels for variables *C* and *D*. If we call the new problem P', then Figure 7.8(c) shows the constraint hypergraph of P' and Figure 7.8(d) shows the constraint graph of the dual problem of P'.

(2) If S_1 and S_2 are sets of variables, and S_1 is a subset of S_2, then the constraint on S_1 can be discarded if we create or tighten an existing constraint on S_2 appropriately. For example, the constraints $C_{C,D}$, $C_{C,E}$ and $C_{D,E}$ in P' in Figure 7.8(c) can be replaced by a new 3-constraint $C_{C,D,E}$ such that all the compound labels and only those compound labels which satisfy all $C_{C,D}$, $C_{C,E}$ and $C_{D,E}$ are put into $C_{C,D,E}$. If we call the new problem after such replacement P'', then Figure 7.8(e) shows the constraint hypergraph of P'' and Figure 7.8(f) shows the constraint graph of the dual problem of P''.

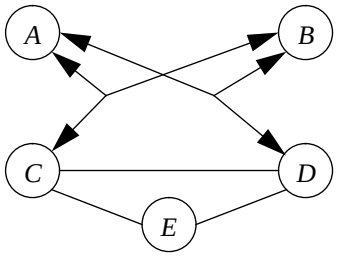
(3) We mentioned that every constraint in the dual problem requires no more than assigning consistent values to the shared variables (in the original problem) in the two constrained variables (in the dual problem). We know that equality is transitive ($A = B$ and $B = C$ implies $A = C$). Therefore, in the constraint graph of a dual problem, an edge (*a*, *b*) is redundant, and therefore can be removed if there exists an alternative path between nodes *a* and *b*, such that $a \cap b$ appears on every edge in the path (*a* and *b* are sets of variables in the original problem). For example, in the constraint graph in Figure 7.8(d), the edge (*ABC*, *CE*) can be removed because *C* is the only shared variable on this edge, and *C* also appears in both of the edges (*ABC*, *CD*) and (*CD*, *CE*) (((*ABC*, *CD*), (*CD*, *CE*)) is a path from *ABC* to *CE*). Alternatively, if the edge (*ABC*, *CE*) is retained, then one of (*ABC*, *CD*) or (*CD*, *CE*) can be removed for the same reason. Similarly, one of the edges (*ABD*, *DE*), (*ABD*, *CD*) or (*CD*, *DE*) is redundant.



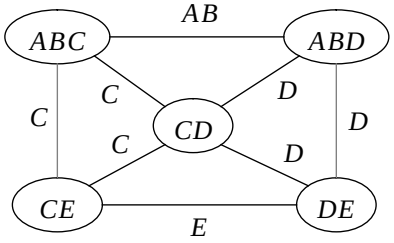
(a) The constraint hypergraph of a CSP P



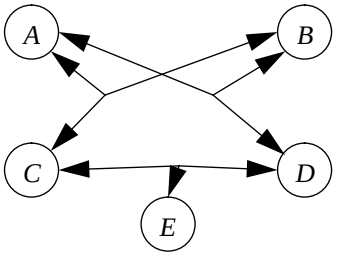
(b) The constraint graph of the dual problem P^{d}; the same values must be taken by the shared variables shown on the edges



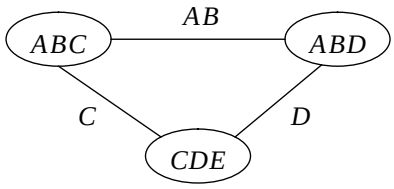
(c) P with redundant constraint (C, D) added to it, which is satisfied by all compound labels for C and D



(d) The constraint graph of the dual problem in (c); the edges (*ABC*, *CE*) and (*ABD*, *DE*) are redundant, and therefore can be removed



(e) Problem in (c) with constraints (C, D), (C, E) and (D, E) replaced by a 3-constraint on (C, D, E)



(f) The constraint graph of the dual problem in (e)

Figure 7.8 Examples of equivalent CSPs and their dual problems

7.6.3 Overview of the tree-clustering method

The general strategy underlying the tree-clustering method can be summarized as in Figure 7.9. Given a CSP P, one can formulate its dual problem P^d, and take each cluster of variables in P^d as a subproblem. A solution for a cluster is a compound label in P. By combining the compound labels for each cluster in P^d, one gets a solution for P.

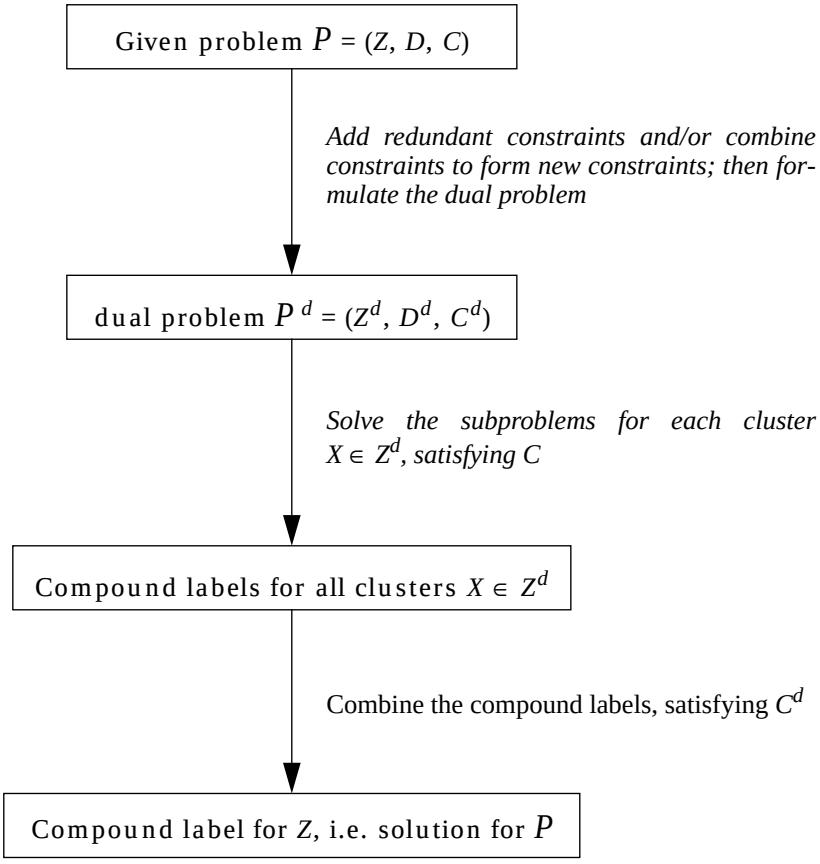


Figure 7.9 General strategy underlying the tree-clustering method

Let n and a be the number of variables and the maximum domain size in P, respectively, and s be the size of the greatest cluster in P^d. Since $s \leq n$, the complexity of solving the subproblems in P^d (which is $O(a^s)$ in general), is no greater than the complexity of solving P (which is $O(a^n)$ in general).

However, there is one serious problem in the compound labels combination step. That is caused by the cycles in the constraint graph of P^d. Consider the constraint graph in Figure 7.8(f). After finding a compound label cl_1 for *ABC* and a compound label cl_2 for *ABD*, there may not be any compound label for *CDE* which is compatible with both cl_1 and cl_2. In the worst case, one has to backtrack through all the compound labels for *ABC* and *ABD* before finding a compatible compound label for *CDE*, or realizing that no solution exists.

Let there be k clusters in P^d, and the number of variables of each cluster be s_1, s_2, ..., s_k. In the worst case, the number of solutions for these clusters, i.e. the domain sizes of the variables in P^d, are $O(a^{s_1})$, $O(a^{s_2})$, ..., $O(a^{s_k})$. Thus, the complexity of the combination step could be $O(a^{s_1} \times a^{s_2} \times ... \times a^{s_k}) = O(a^{s_1 + s_2 + \ldots + s_k})$, which could be higher than $O(a^n)$.

The solution to the combination problem is to make sure that the constraint graph of P^d is a tree. If we succeed in achieving this, then the combination problem can be solved efficiently using the tree-search algorithm described in Section 7.2. In the following we shall explain how, by adding and removing redundant constraints, a general CSP can be transformed into one whose dual problem's constraint graph forms a tree.

Let k be the number of clusters and r be the size of the largest cluster in the dual problem. The largest possible domain size of the variables in the dual problem is therefore $O(a^r)$. The complexity of applying the tree-searching algorithm to the combination problem is then $O(k(a^r)^2)$ (or $O(ka^{2r})$). In fact, if all the compound labels are ordered by the variables lexicographically, finding whether a compound label has a compatible compound label in another variable in P^d requires $O(\log a^r)$ instead of $O(a^r)$. Therefore, the overall complexity of the tree-searching algorithm could be reduced to $O(ka^r \log a^r) = O(kra^r \log a)$. However, when a cluster is constrained by more than one other cluster, more than one ordering may be needed for the compound labels; for example, if the cluster {*A*, *B*} has only three labels ordered as: (<*A*,1><*B*,2>), (<*A*,2><*B*,1>) and (<A,3><*B*,3>), this ordering would help checking the redundancy of compound labels for {*A*, *C*} (because *A* is ordered), but not for {*B*, *D*} (because *B* is not ordered).

The question is how to make sure that the constraint graph of the dual problem forms a tree. The answer to this is provided in the literature on query optimization in database research.[3] The key is to generate acyclic hypergraphs, as explained below.

Definition 7-4:

A **clique** in a graph is a set of nodes which are all adjacent to each other:

$\forall$ graph((V, E)): $\forall\ Q \subseteq V$:
(clique(Q, (V, E)) $\equiv$ ($\forall\ x, y \in\ Q$: $x \neq y \Rightarrow (x, y) \in E$)) ■

Definition 7-5:

A **maximum clique** is a clique which is not a proper subset of any other clique in the same graph:

$\forall$ graph((V, E)): $\forall$ clique(Q, (V, E)):
(maximum_clique(Q, (V, E)) $\equiv$ ($\neg\ \exists\ Q'$: (clique(Q', (V, E)) $\wedge\ Q \subset Q'$)) ■

Definition 7-6:

The **primal graph** G **of a hypergraph** $\mathcal{G}$ is an undirected graph which has the same nodes as the hypergraph, and every two nodes which are joined by any hyperedge in $\mathcal{G}$ is joined by an edge in G. For convenience, we denote the primal graph of $\mathcal{G}$ by G($\mathcal{G}$):

$\forall$ hypergraph(($\mathcal{N}$, $\mathcal{E}$)):
(V, E) = primal_graph(($\mathcal{N}$, $\mathcal{E}$)) $\equiv$
(($V = \mathcal{N}$) $\wedge$ E = { (x, y) | $x, y \in\ \mathcal{N} \wedge (\exists\ e \in \mathcal{E} : x, y \in e)$}) ■

Definition 7-7:

A hypergraph $\mathcal{G}$ is **conformal** if, for every maximum clique in its primal graph, there exists a hyperedge in $\mathcal{G}$ which joins all the nodes in this maximum clique:

$\forall$ hypergraph(($\mathcal{N}$, $\mathcal{E}$)):
(conformal(($\mathcal{N}$, $\mathcal{E}$)) $\equiv$
G = primal_graph(($\mathcal{N}$, $\mathcal{E}$)) $\Rightarrow$

3. Like a CSP, a relational database can be seen as a hypergraph; but this will not be elaborated further here.

$$(\forall\, Q \subseteq N : \text{maximum\_clique}(Q, G) \Rightarrow Q \in E)) \blacksquare$$

Definition 7-8:

A **chord** in an undirected graph is an edge which joins two nodes which are accessible to each other without going through this edge:

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

Definition 7-9:

A graph is **chordal** if every cycle with at least four distinct nodes has an edge joining two nonconsecutive nodes in the cycle (this edge is by definition a chord):

$\forall$ graph((V, E)):
chordal((V, E)) $\equiv$
$\forall\, x_1, x_2, x_3, ..., x_m \in V$:
$(m \geq 4 \wedge (\forall\, x_i, x_j: x_i \neq x_j) \wedge \text{path}((x_1, x_2, x_3, x_m, x_1), (V, E))) \Rightarrow$
$(\exists\, a, b \in \{x_1, x_2, x_3, ..., x_m\}$:
$((a,b) \in E \wedge \neg\, (a,b) \in \{(x_1, x_2), (x_2, x_3), ..., (x_m, x_1)\})\ ))$ $\blacksquare$

Definition 7-10:

A hypergraph is **reduced** if and only if no hyperedge is a proper subset of another:

$\forall$ hypergraph((N, E)):
(reduced-hypergraph((N, E)) $\equiv$ $(\forall\, e \in E: \neg\, (\exists\, e' \in E: e \subseteq e'))$) $\blacksquare$

Combined with Definition 7.7, a hypergraph G is reduced and conformal if and only if every hyperedge in G joins all the nodes in a maximum clique in its primal graph, and every maximum clique in the primal graph is joined by a hyperedge in G.

Definition 7-11:

Given a hypergraph (N, E) and any subset of nodes M, the set of all hyperedges in E with nodes which are not members of M removed (except the hyperedge which joins an empty set of nodes) is called a **node generated set of partial hyperedges**:

$\forall$ hypergraph((N, E)):

(∀ hypergraph((N , F)):
(node-generated-hyperedges(F, (N , E)) ≡
(∃ $M \subseteq N$: F = {$e \cap M \mid e \in E$ } – {{}}))) ■

Definition 7-12:

A **path in a hypergraph** is a sequence of *k* hyperedges, with $k \geq 1$, such that the intersection of adjacent hyperedges are nonempty:

∀ hypergraph((N , E)):
(∀ e_1, e_2, ..., $e_k \in E$:
(path((e_1,e_2,...,e_k), (N , E)) ≡ (∀ $1 \leq i < k$: $e_i \cap e_{i+1} \neq \{\}$))) ■

Definition 7-13:

A hypergraph is **connected** if and only if there exists a path which connects any two nodes:

∀ hypergraph((N , E)):
(connected((N , E)) ≡
(∀ $P, Q \in N$: (∃ e_1, e_2, ..., $e_k \in E$:
($P \in e_1 \wedge Q \in e_k \wedge$ path((e_1,e_2,...,e_k), (N , E)))))) ■

Definition 7-14:

A set of nodes *A* is an **articulation set** of a hypergraph G if it is the intersection of two hyperedges in *G*, and the result of removing *A* from G is a hypergraph which is not connected:

∀ reduced-hypergraph((N , E)): connected((N , E)):
(∀ $A \subseteq N$:
(articulation_set(A, E) ≡
(∃ $e_1, e_2 \in E$: $A = e_1 \cap e_2$) ∧
¬ connected($N - A$, {$e - A \mid e \in E$ } – {{}}))) ■

We continue to use **nodes_of(E)** to denote the set of nodes involved in the hyperedges E of a hypergraph (Definition 1-17):

nodes_of(E) ≡ {$x \mid \exists\, e \in E$: $x \in e$}

Definition 7-15:

A **block** of a reduced-hypergraph is a connected, node-generated set of partial hyperedges with no articulation set:

$\forall$ reduced-hypergraph((N, E)):
  $\forall$ hyperedges(F, N):
    block(F, (N , E)) $\equiv$
      node-generated-hyperedges(F, (N , E)) $\wedge$
      connected((nodes_of(F), F)) $\Rightarrow$
        $\neg$ ($\exists$ $S \subseteq N$: articulation_set(S, F)) ■

Recall in Definition 1-6 that hyperedges(F, N) means that F is a set of hyperedges for the nodes N in a hypergraph.

Definition 7-16:

A reduced-hypergraph is **acyclic** if and only if it does not have blocks of size greater than 2:

$\forall$ reduced-hypergraph((N, E)):
  (acyclic((N, E)) $\equiv$
    $\forall$ F: hyperedges(F, N): block(F, (N , E)) $\Rightarrow$ $|F| \leq 2$) ■

We shall borrow the following theorem from database research. The proof of this theorem is well documented in the literature (e.g. see Beeri *et al*., 1983]; Maier, 1983).

Theorem 7.5

A reduced-hypergraph is acyclic if and only if it is conformal and its primal graph is chordal:

$\forall$ reduced-hypergraph((N, E)):
  (acyclic((N, E)) $\Leftrightarrow$ conformal((N, E)) $\wedge$ chordal(G((N, E)))

Proof

(see Beeri *et al*. [1983])

The main implication of Theorem 7.5 is that by transforming the CSP to an equivalent problem which constraint hypergraph is conformal, and which primal graph is

chordal, one can ensure that the Tree_search algorithm can be applied in the combination step. The steps of the tree-clustering method in Figure 7.9 are thus refined in Figure 7.10.

In the following two sections, we shall explain how to generate a chordal and conformal CSP which is equivalent to any given CSP. Then we shall introduce a procedure which employs the tree-clustering method.

7.6.4 Generating chordal primal graphs

This section introduces an algorithm for generating chordal primal graphs. Given a graph, chordality is maintained by adding extra edges into it whenever necessary. The basic algorithm is to give the nodes of the graph an ordering, and then process them one at a time. When a node *x* is processed, it is joined to any other node which is (a) before *x* in the ordering; (b) sharing a common parent with *x*; and (c) not already adjacent to *x*. The *Fill_in-1* procedure is a naive implementation of this algorithm:

```
PROCEDURE Fill_in-1((V, E))
/* given a graph (V, E), return a chordal graph with possibly added
   edges */
BEGIN
   /* initialization */
   Ordering ← Max_cardinality_ordering(V, E);
   N ← number of nodes in V;
   /* achieving chordality, by possibly adding extra edges */
   FOR i = 1 to N DO
      FOR j = 1 to i DO
         IF (Ordering[i] and Ordering[j] have common parent)
         THEN IF ((Ordering[i], Ordering[j]) is not already in E)
                  THEN E ← E + {(Ordering[i], Ordering[j])};
   return((V, E));
END /* of Fill_in-1 */
```

The Fill_in-1 procedure will generate a chordal graph no matter what ordering is being used in the initialization. The maximum cardinality ordering (described in Chapter 6) is used because it can be shown that when the graph is already chordal, no addition of edges will be generated by the above algorithm if the maximum cardinality ordering is used [TarYan84]. (Nodes may be added even when the graph is chordal when this algorithm uses some other orderings.)

If the neighbourhood of every node is stored by a bit pattern, then testing whether two nodes have the same parents in an ordering takes roughly a constant time. In

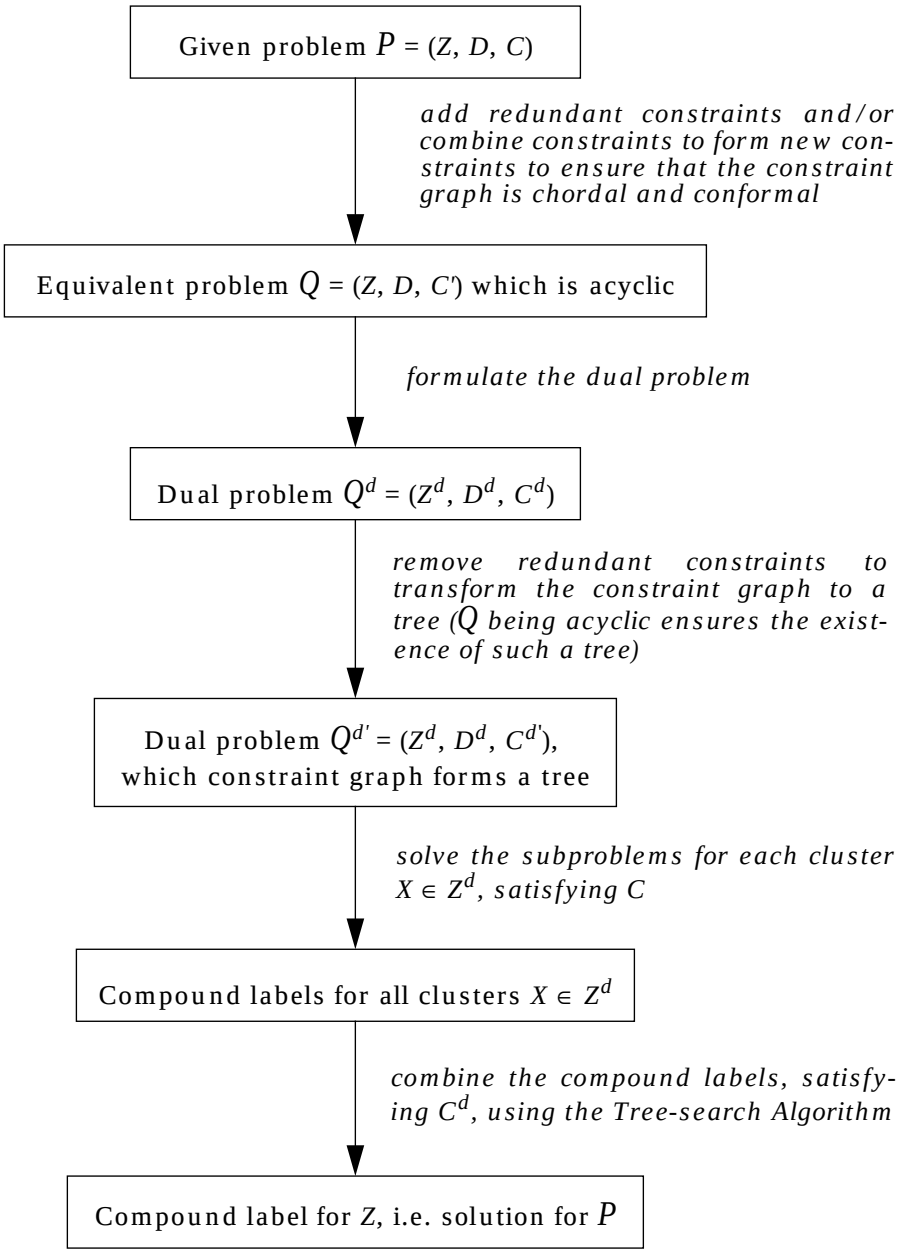


Figure 7.10 Conceptual steps of the tree-clustering method (note that one need not actually construct Q in an implementation)

this case, the procedure Fill_in-1 takes $O(n^2)$ time to complete, because it examines every combination of two nodes in its two FOR loops.

By using more complex data structures, the Fill_in-1 procedure can be improved to run in $O(m+n)$ time, where m is the number of arcs and n is the number of nodes in the graph. For simplicity, without affecting the results of our analysis of the complexity of the tree-clustering method, interested readers are referred to Tarjan & Yannakakis [1984] for improvement of Fill_in-1.

Figure 7.11 shows an example of a constraint graph, and summarizes the procedure for maintaining chordality in the graph.

The ordering (G, F, E, D, C, B, A) is one maximum cardinality ordering for the given graph. The edge (C, D) is added because they are both adjacent to and after the node E. Similarly, the edge (A, E) is added because they are both adjacent to and after the node F.

7.6.5 Finding maximum cliques

By adding necessary redundant constraints using the Fill_in-1 procedure, the constraint graph is made chordal. In order to make the constraint hypergraph of a CSP conformal, we need to identify the maximum cliques in the primal graph (so that we can create a constraint for each maximum clique). In this section, we shall first present a general algorithm for finding maximum cliques. Then we shall present a more efficient algorithm which can be applied after running the Fill_in-1 procedure.

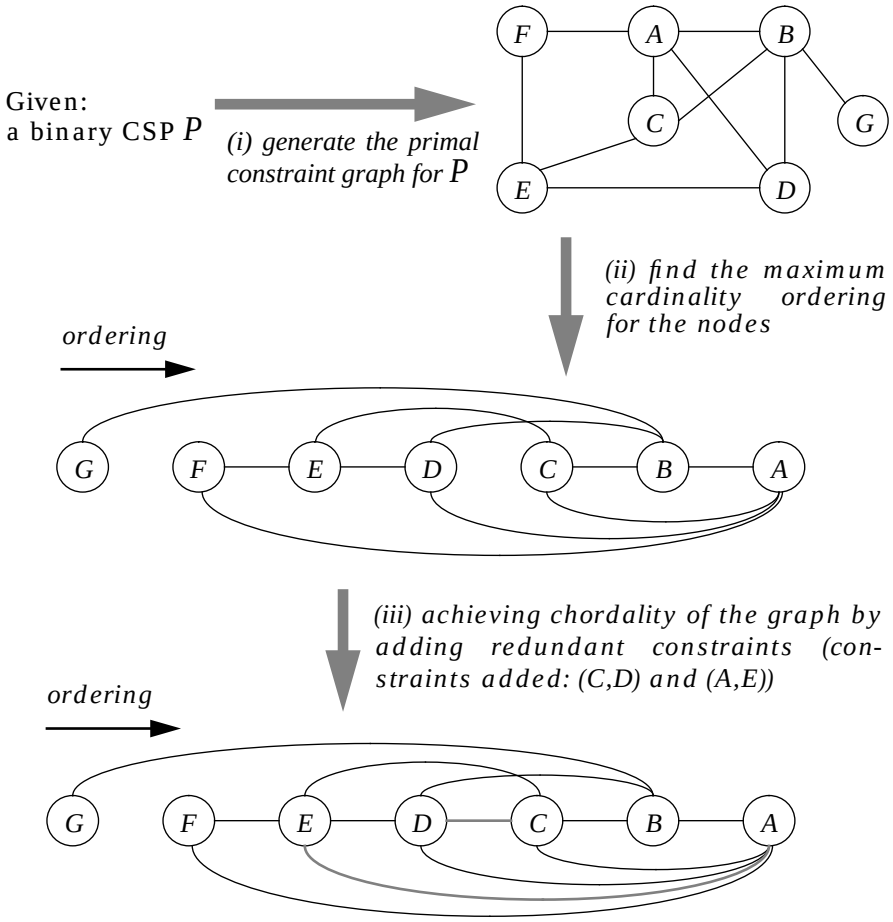
7.6.5.1 A general algorithm for finding maximum cliques

In this section, a general algorithm for finding maximum cliques is introduced. It is based on two observations:

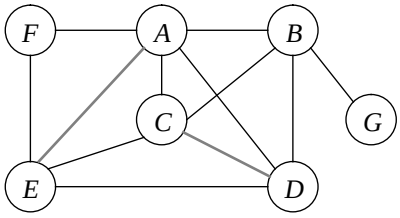
(a) If x is a node in a maximum clique C in a graph, then C must contain x and its neighbours only. (This is trivially true.)

(b) If S is a set of nodes in a graph, and every node in S is adjacent to some node x which is not in S, then S does not contain any maximum clique. This is because if there exists a clique C in S, then $C + \{x\}$ must be a clique (as $C + \{x\}$ forms a complete sub-graph in the given graph). Hence C cannot be a maximum clique (as it is a proper subset of $C + \{x\}$).

Based on these observations, the *Max_cliques-1* procedure finds maximum cliques in a given graph by performing a binary search. In this procedure, one node is considered at a time. One branch of the search looks for maximum cliques which include this node, and the other branch looks for maximum cliques which do not include this node:



(a) Procedure for achieving chordality



(b) A chordal constraint graph generated from the problem P in (a)

Figure 7.11 Example showing the procedure of chordal graphs generation

```
PROCEDURE Max_cliques-1((V, E))
/* given a graph (V, E), returns the set of all maximum cliques */
BEGIN
    Maximum_cliques ← MC(V, E, { });
    return(Maximum_cliques);
END /* of Max_cliques-1 */


PROCEDURE MC(V, E, N)
/* V is a set of nodes; E is a set of edges which may join nodes other
    than those in V; N is a set of nodes which are not in any maximum
    clique */
BEGIN
    IF No_cliques(V, E, N) THEN return({ })
    ELSE IF (is_clique(V, E)) THEN return({V})
            /* is_clique is explained in text */
            ELSE BEGIN
                    x ← any node from V;
                    /* find cliques which contain x */
                    V' ← {x} + set of nodes in V adjacent to x – N;
                    MC1 ← MC(V', E, N);
                    /* find cliques which do not contain x */
                    MC2 ← MC(V – {x}, E, N + {x});
                    return(MC1 + MC2);   /* return all cliques found */
                 END
END /* of MC */


PROCEDURE No_cliques(V, E, N)
/* based on observation (b), that if there exists a node outside V
    which is adjacent to every node in V, then no maximum clique
    exists in V */
BEGIN
    FOR each x in N DO
        IF (x is adjacent to all nodes in V with regard to E)
        THEN return(True);
    return(False);
END /* of No_cliques */
```

The *is_clique(V, E)* procedure checks to see if every pair of nodes in *V* are joined by an edge in *E*. We assume that by using an appropriate data structure (e.g. recording the adjacency of the nodes by bit patterns), *is_clique* can be implemented in *O(n)*, where *n* is the number of nodes in the graph. Besides, since one node is considered at a time, the recursive call of *MC* is at most *n* levels deep. So the overall time com-

plexity of Max_clique is $O(2^n)$.

Figure 7.12 shows the steps of finding the maximum cliques in an example graph. The maximum cliques found are: {*A, B, C, D*}, {*A, C, D, E*}, {*A, E, F*} and {*B, G*}. The sets {*A, B, C, D*} and {*A, C, D, E*} are accepted as maximum cliques because they are complete graphs (by definition of maximum cliques). The complete graph which contains {*A, D, E*} is rejected because all its nodes are adjacent to *B*, which is excluded as an element of any maximum cliques under that branch of the search tree. The fact that node *G* is considered after *A* and *B* on the right most branch of Figure 7.12 is just a convenience for presentation. (If other nodes are considered instead, the search would be deeper, though the result would be the same.)

The efficiency of the construction of *MC2* in the Max_clique algorithm can be improved through the reduction of the size of the remaining graph (call it ***G***). When looking for maximum cliques which do not contain *x*, one can do more than removing *x* from ***G***: one can also remove any neighbour *y* of *x* such that *y*'s neighbourhood is a subset of *x*'s neighbourhood plus *x*:

$$(\{y\} + \text{neighbourhood}(y, \boldsymbol{G})) \subseteq (\{x\} + \text{neighbourhood}(x, \boldsymbol{G}))$$

This is in fact a lookahead step, because if *y* is in any clique, then this clique must contain *y* and nodes in its neighbourhood only. If this clique is a subset of *x* plus its neighbourhood, then this clique cannot be a maximum clique.

For example, in Figure 7.12, when node *A* is excluded from the cliques (i.e. the top right hand side branch), node *D* could have been excluded as well, because {*D*} + neighbourhood(*D*, ***G***) is {*A, B, C, E*}, which is a subset of {*A*} + neighbourhood(*A*, ***G***), which is {*A, B, C, D, E, F*}. The search indeed confirms that *D* does not appear in any maximum clique under that branch of the search tree. By the same token, nodes *C*, *E* and *F* could have been removed when *A* is removed.

Program 7.3, *max-clique.plg*, shows a Prolog implementation of the Max_clique algorithm.

7.6.5.2 Finding maximum cliques after Fill_in-1

Observe that Fill_in-1 gives the nodes in the input graph a total ordering. If this ordering is made accessible to other procedures after the exit of Fill_in-1, then it can help us to find the maximum cliques in the chordal graph efficiently. Fill_in-1 makes sure that for every node *x*, all the children of *x* (according to the given ordering) are connected to each other. This means that *x* and all its children together must be a clique. To find all the maximum cliques in the chordal graph, all one needs to do is to go through the nodes according to this ordering and check whether the clique formed by the focal node and its children is maximum. The pseudo code of this algorithm is shown below:

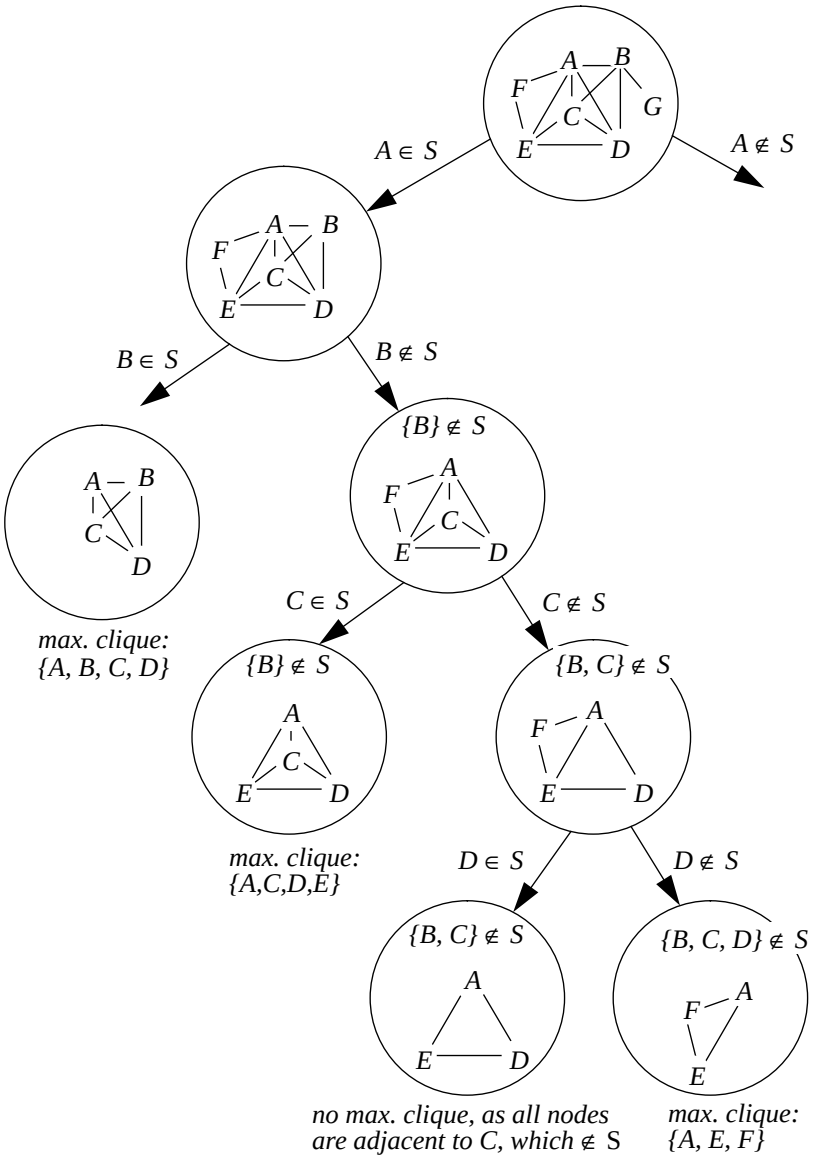


Figure 7.12 Example showing the space searched in identifying the maximum cliques; the set of maximum cliques found are: {*A*, *B*, *C*, *D*}, {*A*, *C*, *D*, *E*}, {*A*, *E*, *F*} and {*B*, *G*}

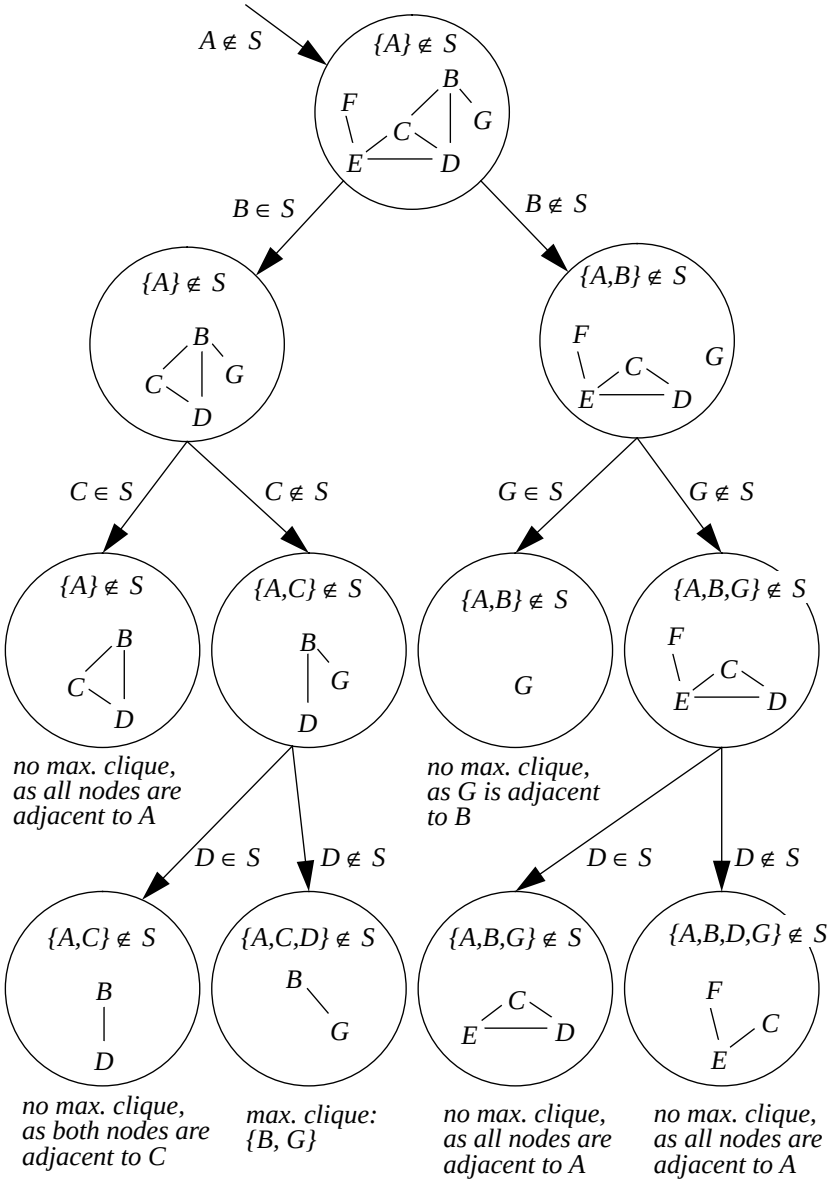


Figure 7.12 (Cont'd) Example showing the space searched in identifying the maximum cliques

```
PROCEDURE Max_cliques-2((V, E), Ordering)
/* (V, E) is a chordal graph generated by Fill_in-1; Ordering is an
    array of nodes V used by Fill_in-1 in generating (V, E) */
BEGIN
    C ← { };      /* C stores the set of maximum cliques found so far */
    FOR i = 1 to n DO     /* n = number of nodes in the input graph */
        BEGIN
            S ← {Ordering[i]} + neighbourhood( Ordering[i], (V, E) );
            IF (S is not a subset of any element in C);
            THEN C ← C + {S};       /* S is a maximum clique */
            V ← V – Ordering[i];
            E ← E –  all edges joining  Ordering[i];
        END
    return(C);
END /* of Max_cliques-2 */
```

Let n be the number of nodes in the input graph. The FOR loop in Max_cliques-2 iterates exactly n times. At most one maximum clique is added to C in each iteration. Therefore, the size of C is at most n. If it takes a constant time to check whether a set is a subset of another, then the IF statement in the FOR loop takes $O(n)$ time. Therefore, the worst case time complexity of Max_cliques-2 is $O(n^2)$. If every set takes $O(n)$ space to store, then the space complexity of Max_cliques-2 is also $O(n^2)$.

Table 7.1 Cliques and maximum cliques in the chordal graph in Figure 7.11

Ordering	Focal Node	Clique	Analysis
1	*G*	{*G, B*}	this is a maximum clique
2	*F*	{*F, E, A*}	this is a maximum clique
3	*E*	{*E, D, C, A*}	this is a maximum clique
4	*D*	{*D, C, B, A*}	this is a maximum clique
5	*C*	{*C, B, A*}	this is not a maximum clique, as it is a subset of {*D, C, B, A*}
6	*B*	{*B, A*}	this is not a maximum clique, as it is a subset of {*D, C, B, A*}
7	*A*	{*A*}	this is not a maximum clique, as it is a subset of {*D, C, B, A*}

Table 7.1 shows the cliques and maximum cliques of the chordal graph shown in Figure 7.11(b). This example illustrates that the procedure Max_cliques-2 finds the same maximum cliques as Max_cliques-1.

7.6.6 Forming join-trees

Recall that given a CSP P = (*Z*, *D*, *C*), each variable of its dual problem P^d is a set of variables in *Z*, and its domain being a compound label for the set of variables in P. Binary constraints and binary constraints only exist in P^d. A binary constraint exists between two variable in P^d if they share some common variables in P. Point (3) in Section 7.6.2 explains that since all constraints concern about equality which is transitive, redundant constraints can be removed trivially.

Results in the graph theory literature show that given a CSP whose constraint hypergraph is acyclic, the constraint graph of its dual problem can be reduced (by removing redundant constraints) to a tree. Such a tree is called a **join-tree**. In the preceding sections, we have explained how to transform a CSP to one which constraint hypergraph is acyclic. This section explains how join-trees can be constructed for dual problems. Again, we shall first present a general algorithm for finding join-trees, then we present an algorithm which makes use of the ordering produced by Fill_in-1.

7.6.6.1 General algorithm for finding join-trees

The following is the pseudo code for an algorithm to establish the constraints for a given set of hyperedges. It is modified from *Graham's Algorithm*, which is used to determine whether a hypergraph is acyclic (see Beeri *et al*., 1983]).

```
PROCEDURE Establish_constraints-1(MC)
/* MC is a set of hyperedges in a hypergraph; this hypergraph must
   be acyclic; otherwise this procedure will never terminate! */
BEGIN
   C ← { };        /* C is to be returned as a set of constraints on MC */
   index elements in MC with numbers 1 to k;
   S ← MC together with the indices;
        /* S[i] = MC[i] = the i-th maximum clique */
   /* manipulate the elements in S in order to establish links in MC */
   WHILE (S ≠ { }) DO
      BEGIN
         FOR i = 1 to k DO
            FOR each variable x in S[i] DO
               IF (x does not appear in any S[j] where j ≠ i)
```

```
            THEN S[i] ← S[i] – {x};
    FOR i = 1 to k DO
        IF (there exists some S[j] in S, where  j ≠ i and S[i] ⊆ S[j])
        THEN BEGIN
                    C ← C + C_MC[i],MC[j], where C_MC[i],MC[j] is a
                        constraint which requires consistent
                        labelling to MC[i] and MC[j];
                    S ← S – S[i];
                END
    END;   /* MC being acyclic guarantees termination of WHILE*/
return(C);
END /* of Establish_constraints-1 */
```

The Establish_constraints-1 procedure basically repeats the following steps:

(i) remove any variable which appears in one hyperedge only;

(ii) link hyperedges *S*[*i*] and *S*[*j*] if *S*[*i*] is a subset of *S*[*j*]; remove *S*[*i*] from the set of hyperedges.

If the input *MC* forms the edges of an acyclic hypergraph, then *S* will always be reduced to an empty set, and the procedure will terminate (see Beeri *et al*., 1983).

In the worst case, each of the two out-most FOR loops needs to consider every pairs of *S*[*i*] and *S*[*j*]. Therefore, the complexity for both of them are $O(k^2)$, where *k* is the size of *MC* (i.e. the number of hyperedges). In the worst case, only one element is removed from *S*. When this is the case, the WHILE loop will have to iterate *k* times to eliminate all the elements in *MC*. Therefore, the overall worst case complexity of the algorithm Establish_constraints-1 is $O(k^3)$.

7.6.6.2 Finding join-trees after Fill_in-1 and Max_cliques-2

If the maximum cliques are returned by Max_cliques-2 following Fill_in-1, then one can build the join-tree more efficiently than Establish_constraints-1. Again, the total ordering of the nodes in the primal graph which is used in Fill_in-1 must be made accessible. Let us call this ordering <. A little reflection should convince readers that Max_cliques-2 ensures that a node can only have the highest precedence according to < in, at most, one of these maximum cliques. Therefore, the maximum cliques can be ordered according to the ordering of their nodes which have the highest precedence according to <. Results in the graph theory literature suggest that, given such an ordering, one can create the join-tree by simply connecting every maximum clique *mc* to a maximum clique which is (a) after *mc* according to this ordering, and (b) shares the maximum number of nodes with *mc* . The pseudo code of this algorithm is shown below:

```
PROCEDURE Establish_constraints-2(MC, Ordering)
/* MC is a set of maximum cliques of the primal graph or a CSP */
/* MC must be returned by Max_cliques-2 */
/* Ordering is a total ordering of the variables of the CSP returned by
   Fill_in-1 */
BEGIN
   C ← { };          /* C = set of constraints on MC established so far*/
   Order the sets in MC according to the Ordering of their earliest
      elements;
   FOR i = 1 to |MC| – 1 DO
      /* join MC[i] to the MC[k] (i < k) which shares the maximum
         number of elements with it */
      BEGIN
         MNSN ← 0;      /* MNSN = max. number of shared nodes */
         FOR j = i + 1 to |MC| DO
            IF(|(MC[i] ∩ MC[j])| > MNSN)
            THEN BEGIN
                              MNSN ← |MC[i] ∩ MC[j])| ; k ← j;
                         END;
         C ← C + C_MC[i],MC[k], where C_MC[i],MC[k] is a constraint
            which requires consistent labelling to MC[i] and MC[k];
      END
   return(C);
END /* of Establish_constraints-2 */
```

Let k be the number of maximum cliques in MC. If set intersection takes a constant time, then it takes $O(k)$ time to find the maximum clique which shares the maximum number of nodes of a particular maximum clique. The two FOR loops together dominate the worst case time complexity of Establish_constraints-2, which is $O(k^2)$.

Table 7.2 Join-tree for the maximum cliques found in Table 7.1

Ordering	Maximum clique	Maximum clique of lower ordering which shares the maximum elements	Constraint Created
1	{*G*, *B*}	{*D*, *C*, *B*, *A*}	$C_{GB,DCBA}$
2	{*F*, *E*, *A*}	{*E*, *D*, *C*, *A*}	$C_{FEA,EDCA}$
3	{*E*, *D*, *C*, *A*}	{*D*, *C*, *B*, *A*}	$C_{EDCA,DCBA}$
4	{*D*, *C*, *B*, *A*}		(root)

Going back to the above example, the maximum cliques, their ordering and the constraints generated are shown in Table 7.2. The join-tree thus created is shown in Figure 7.13.

7.6.7 The tree-clustering procedure

The Tree_clustering procedure, which makes use of the procedures introduced so far, implements the tree-clustering method:

```
PROCEDURE Tree_clustering(Z, D, C)
BEGIN
    GG ← hypergraph of (Z, D, C);
    G ← primal_graph of GG;
    G ← Fill_in-1(G);                    /* generate chordal primal graph */
    /* we assume that Ordering is produced by Fill_in-1 as a side
       effect */
    MC ← Max_cliques-2(G, Ordering);     /* identify max. cliques */
    D^d ← { };
    FOR each mc ∈ MC DO                  /* solve one sub-problem */
        BEGIN
            D_mc ← {D_x | x ∈ mc ∧ D_x ∈ D};      /* specifify domains */
            T_mc ← solution tuples for the CSP (mc, D_mc, CE(mc, (Z, D,
                C)));
            D^d ← D^d + {T_mc};
        END
    C^d ← Establish_constraints-2(MC, Ordering);
    S^d ← Tree_search(MC, D^d, C^d);
    /* S^d is a solution to the dual problem, i.e. a set of compound
       labels for the original problem which assigns a unique value to
       each variable in Z */
    Solution ← (<x1,v1><x2,v2>...<xn,vn>) where {x1, x2, ..., xn} = Z
       and for all 1 ≤ i ≤ n, (<xi,vi>) is the projection of some com-
       pound labels in S^d;
    /* Solution is a compound label to the original problem (Z, D, C) */
    return(Solution);
END /* of Tree_clustering */
```

The Fill_in-1 procedure adds redundant constraints into the graph to make it chordal. The Max_cliques-2 procedure returns the set of maximum cliques in the graph. The Establish_constraints-2 procedure generates a join-tree for the dual

problem. Each of the edges in this join-tree represents a constraint which requires consistent values to be assigned to the common variables in the joined clusters.

The Tree_clustering procedure adopts the basic ideas explained in Figure 7.10. It first makes sure that the primal graph of the given problem is chordal. Then the maximum cliques are identified. Each maximum clique forms a sub-CSP which will be solved separately. The solution for each maximum clique becomes a constraint on the variables in this maximum clique, replacing the set of all relevant constraints in the original problem; hence the transformed CSP becomes conformal. This ensures that the constraint graph of the dual problem forms a tree. Then the Tree_-search procedure is applied to solve the dual problem. The solution of the dual problem can be used to generate a solution for the original problem quite trivially.

The time complexity of the Fill_in-1 and Max_cliques-2 are both $O(n^2)$, where n is the number of variables in the given CSP. Finding solution tuples for the clusters requires $O(a^r)$ time in general, where a is the maximum domain size of the variables in the given CSP, and r is the size of the largest cluster. Let k be the number of maximum cliques in the transformed CSP. The number of variables in the dual problem is then k. According to the analysis in the last section, the Establish_constraints-2 procedure takes $O(k^2)$ time to complete. The domains of the variables in the dual problem is a^r in the worst case, so the worst case time complexity of the Tree_-search procedure is $O(ka^{2r})$. It can be reduced to $O(ka^r \log (a^r)))$, or $O(kra^r \log (a))$, if the procedure for maintaining DAC can be optimized in the way described above (Section 7.6.3).

The time complexity of Tree-searching, $O(ka^{2r})$, should dominate the time complexity of the Tree_clustering algorithm, because compared with it, n^2, a^r and k^2 (the complexity of Fill_in-1, Max_cliques-2, solving the decomposed problems and Establish_constraints-2) are insignificant.

The example in Figure 7.13 summarizes the steps of the tree-clustering method.

7.7 *j*-width and Backtrack-bounded Search

Theorem 6.1 states the relationship between k-consistency in a CSP and the width of its graph. In this section, we extend the concept of width to j-width, and show that it has interesting results related to (i, j)-consistency.

7.7.1 Definition of *j*-width

In Chapter 2, we defined the concept of backtrack-free search (Definition 2-12). Here, we define a related concept called b-level backtrack-bounded.

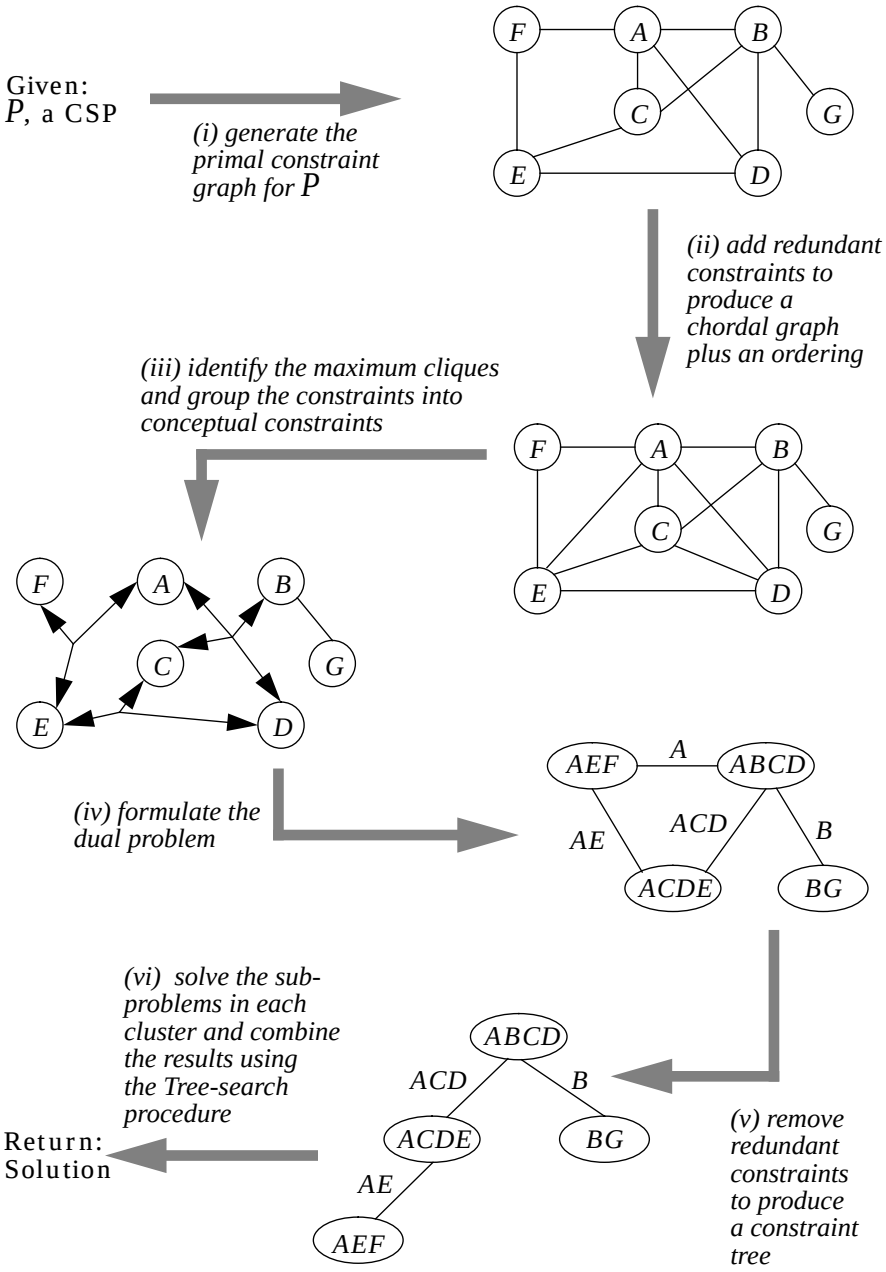


Figure 7.13 Example summarizing the tree-clustering procedure

Definition 7-17:

A backtracking search for solutions in a CSP is called ***b*-level backtrack-bounded**, or ***b*-bounded** for simplicity, under an ordering if, after labelling h variables for any h less than the number of variables in the problem, we can always find a value for the $(h + 1)$-th variable without reconsidering more than the last $b - 1$ labels:[4]

$\forall$ csp((*Z*, *D*, *C*)): ($\forall$ <: total_ordering(*Z*, <): ($\forall$ $b < |Z|$:
b-level-backtrack-bounded((*Z*, *D*, *C*), <) $\equiv$
($\forall$ $x_1, x_2, ..., x_h \in Z$: ($x_1 < x_2 < ... < x_h \Rightarrow$
($\forall$ $v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_h \in D_{x_h}$:
(satisfies((<x_1,v_1>... <x_h,v_h>), CE({x_1, ..., x_h}, (*Z*, *D*, *C*))) $\Rightarrow$
($\forall$ $x_{h+1} \in Z$: ($x_h < x_{h+1} \Rightarrow$
$\exists$ $v'_{h-b+1} \in D_{x_{h-b+1}}, ..., v'_h \in D_{x_h}, v_{h+1} \in D_{x_{h+1}}$:
satisfies((<x_1,v_1> ... <x_{h-b},v_{h-b}><x_{h-b+1},v'_{h-b+1}> ... <x_h,v'_h> <x_{h+1},v_{h+1}>), CE({x_1, ..., x_h, x_{h+1}}, (*Z*, *D*, *C*))) ■

In other words, in a chronological backtracking search where *b*-bounded is guaranteed, if one can successfully label h variables without violating any constraints, then one can freeze the first $(h - b)$ labels in labelling the rest of the variables. A backtrack-free search is 0-bounded by definition.

Now we shall look at situations under which searches are *b*-bounded. First, we shall extend the concepts of width for nodes, orderings and graphs in Chapter 3 (see Definitions 3-20 to 3-22) to the width of a sequence of variables in a graph.

Definition 7-18:

The **width of a group of *j* consecutive nodes** in a graph under an ordering is the number of nodes preceding this group which are joined to any of the j nodes in it:

$\forall$ graph((*V*, *E*)): ($\forall$ <: total_ordering(*V*, <):
($\forall$ $x_1, x_2, ..., x_j \in V$: consecutive(($x_1, x_2, ..., x_j$)):
(width(($x_1, x_2, ..., x_j$), (*V*, *E*), <) $\equiv$

4. Note that the *b* in the definition of *b-level-backtrack-bounded*, or *b-bounded*, is actually treated as an argument of the predicate (like the *k* in *k-consistency* in Chapter 3). A more accurate syntax which conforms to first order logic would be to put *b* between the brackets, which makes *b-level-backtack-bound(b, Compound_label, C_s)*. The present syntax is adopted for both simplicity and conformation with the CSP literature. The same arrangement applies to the definition of *j*-width later in this chapter.

$$\left| \{z \mid z \in V \wedge z \notin \{x_1, x_2, ..., x_j\} \wedge \exists\, w\colon (w \in \{x_1, x_2, ..., x_j\} \wedge z < w \wedge (z, w) \in E)\} \right|)))$$

where

$$\forall\ \text{graph}((V, E))\colon (\forall <\colon \text{total\_ordering}(V, <)\colon (\forall\ x_1, x_2, ..., x_j \in V\colon \text{consecutive}((x_1, x_2, ..., x_j)) \equiv x_1 < x_2, ..., x_{j-1} < x_j \wedge (\forall\ y \in V\colon (\exists\ 1 \le i \le j\colon y < x_i) \Rightarrow y < x_1) \wedge (\forall\ z \in V\colon (\exists\ 1 \le i \le j\colon x_i < z) \Rightarrow x_j < z))) \blacksquare$$

Definition 7-19:

The ***j*-width of a node** x is the minimum of the widths of all the groups of j or less consecutive nodes which end with x:

$$\forall\ \text{graph}((V, E))\colon (\forall <\colon \text{total\_ordering}(V, <)\colon (\forall\ x_m \in V\colon j\text{-width}(x_m, (V, E), <) \equiv \text{MIN width}((x_{m-k+1}, ..., x_{m-1}, x_m), (V, E), <)\colon 1 \le k \le j)) \blacksquare$$

The concept j-width is a generalization of the concept width. According to this definition, the definition of *the width of a node* in Chapter 3 (Definition 3-20) is equivalent to the *1-width of a node*.

Definition 7-20:

The ***j*-width of a graph under an ordering** is the maximum j-width of all the nodes in the graph under that ordering:

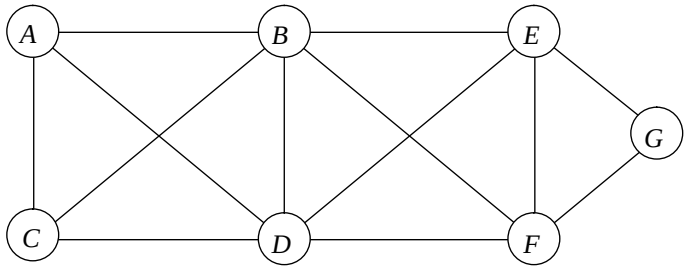
$$\forall\ \text{graph}((V, E))\colon (\forall <\colon \text{total\_ordering}(V, <)\colon j\text{-width}(V, E), <) \equiv \text{MAX } j\text{-width}(x, (V, E), <)\colon x \in V \blacksquare$$

Definition 7-21:

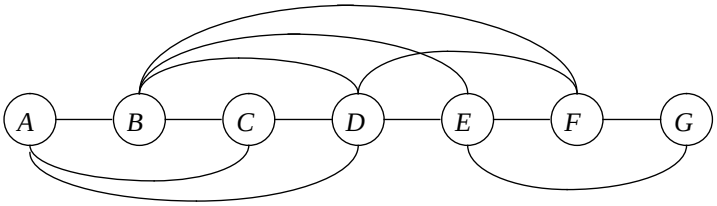
The ***j*-width of a graph** is the minimum j-width of the graph under all possible orderings of its nodes:

$$\forall\ \text{graph}((V, E))\colon j\text{-width}((V, E)) \equiv \text{MIN } j\text{-width}((V, E), <)\colon \text{total\_ordering}(V, <) \blacksquare$$

Figure 7.14 gives an example of a graph and the j-width of the nodes for j's between 1 and 3. For example, the 2-width of node F is 2 because although F is adjacent to three predecessors (B, D and E), E and F together are adjacent to only 2 predecessors (B and D), and the 2-width of F is the minimum of 3 and 2. The 2-width of the ordering shown in Figure 7.14 is the maximum of the j-widths for all the nodes, which is 2.



(a) Example of a graph



1-width:	*0*	*1*	*2*	*3*	*2*	*3*	*2*	**3**
2-width:	*0*	*0*	*1*	*2*	*2*	*2*	*2*	**2**
3-width:	*0*	*0*	*0*	*1*	*2*	*2*	*2*	**2**

(b) j-width of the nodes for $1 \leq j \leq 3$, given the ordering A, B, C, D, E, F, G (the width of each node is shown in italic, the j-width of the orderings are indicated in bold)

Figure 7.14 Example of a graph and the j-widths of an ordering

7.7.2 Achieving backtrack-bounded search

Under the above definitions and the definition for strong (i, j)-consistency in Chapter 3, Freuder [1985] proves the following theorem:

Theorem 7.6 (due to Freuder, 1985)

Given a constraint graph for a CSP, there exists a search order that guarantees j-bounded backtrack search if the graph is strong (i, j)-consistent for i equals to the j-width of the graph.

$\forall$ csp((*Z*, *D*, *C*)): ($\forall$ *i*, *j* $\leq$ | *Z* | :
(strong (*i*, *j*)-consistent(*Z*, *D*, *C*) $\Rightarrow$
($\forall$<: total_ordering(*Z*, <):
(*i* = *j*-width(G((*Z*, *D*, *C*)), <) $\Rightarrow$
j-level-backtrack-bounded((*Z*, *D*, *C*), <)))))

Proof

The proof follows from the definitions. Given a CSP, assume that there exists an ordering whose *j*-width is *i*. Assume further that the problem is strong (*i*, *j*)-consistent. When *x* is the next variable to be labelled, there must exist a $k \leq j$ such that the sequence of *k* variables up to and including *x* has width *i'*, where $i' \leq i$. In other words, this sequence of variables are joined to *i'* variables before this sequence. Given the fact that the problem is strong (*i*, *j*)-consistent, once those *i'* variables are labelled, there exists a legal compound label for these *k* variables which is compatible with the compound label for the *i'* variables. So to assign a value to *x*, one needs to revise no more than the *k* variables before it. Since $k \leq j$, the search is *j*-bounded.

(Q.E.D.)

In other words, given a problem whose *j*-width is equal to *i*, one can determine the bound for one's backtrack search if one can maintain strong (*i*, *j*)-consistency for this problem. This implies that by finding an ordering which has the minimum *j*-width, one can minimize *i* in maintaining strong (*i*, *j*)-consistency.

Freuder points out that the *j*-width of an ordered CSP can be determined by a branch and bound method. Unfortunately, maintaining strong (*i*, *j*)-consistency may change the width of the constraint graph. Besides, there are no efficient algorithms for determining the *j*-width of an ordered CSP and maintaining (*i*, *j*)-consistency. So, although Theorem 7.6 is an interesting observation, its practical use in CSPs solving is yet to be explored.

7.8 CSPs with Binary Numerical Constraints

When all the variables in a CSP are numerical variables, and there exist unary and binary linear constraints only, specialized linear programming techniques can be applied. When variables are allowed to take numbers as their values, the problem is a non-standard CSP (refer to Definition 1-12), because the domains are infinite. However, since the constraints take special forms, efficient algorithms exist for finding solutions for them.

7.8.1 Motivation

Research in such problems is partly motivated by point-based temporal reasoning. In point-based temporal reasoning, time points are taken as primitive objects. Intervals may be represented by pairs of points. One of the tasks in temporal reasoning is to assign a numerical value to each time point, satisfying constraints on them. Simple temporal constraints are:

(a) boundary constraints:
The value of a time point may be given a lower bound, which is called the *earliest time*, and a upper bound, which is called the *latest time*. In other words, given a time point x, there may be constants a and b such that

$$x > a$$
and $$x < b$$

must hold, where < and > can also be ≤ and ≥.

(b) distance constraints:
A *distance constraint* requires that the distance between two points be bounded within a range. For example, if x and y are variables representing two time points, and a and b are constants, a distance constraint may take the following form:

$$x - y > a\ ;$$
$$x - y < b\ ;$$
$$a < x - y < b$$

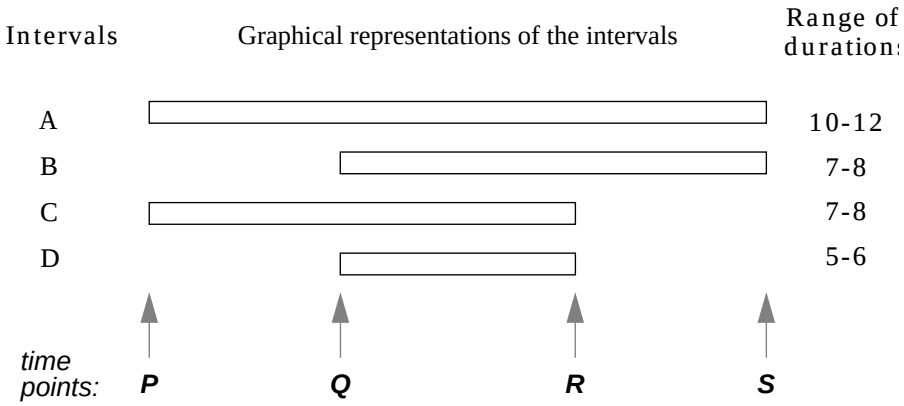
where < and > can also be ≤ and ≥. Examples of distance constraints are upper bounds and lower bounds on *durations*. A *precedence constraint* is a special kind of distance constraint where the constants are 0. In other words, precedence constraints take the form:

$$x < y$$

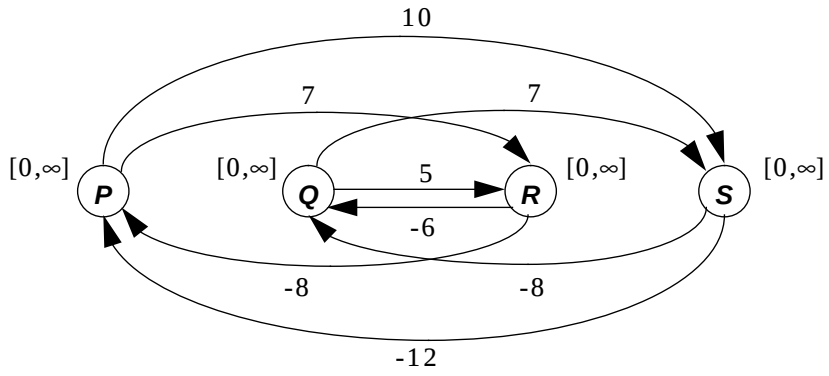
Figure 7.15(a) shows an example of a set of intervals and constraints on them. Intervals here are represented by pairs of time points. Intervals A, B, C and D are represented by (P, S), (Q, S), (P, R) and (Q, R) respectively (for our purpose here, we do not have to worry about the "open" and "closeness" of intervals. Interested readers may refer to van Benthem, 1983). Figure 7.15(a) shows that intervals A and C must start at the same time, C and D must end at the same time, etc. Besides, the durations are constrained to be within bounds:

$$10 \le S - P \le 12$$
$$7 \le S - Q \le 8$$
$$7 \le R - P \le 8$$
$$5 \le R - Q \le 6$$

The situation in Figure 7.15(a) can be represented by a **temporal constraint graph**, a directed graph in which the nodes represent the time points, and the arcs represent



(a) Example set of intervals, points and constraints (assuming no unary constraints on the time points)



(b) Temporal constraint graph of the time points in (a) (the lower- and upper-bounds of the nodes are shown in [ ]'s)

Figure 7.15 Example of a set of constrained intervals and points and their corresponding temporal constraint graph

precedence. Each node is labelled by two numerical values: a lower bound and a upper bound. If the lower and upper bounds are unknown, then 0 and infinity are used, respectively. Each arc is labelled by a numerical value which indicates the distance between the joined time points. If the distance between points *x* and *y* is at least *a*, then an arc is created from *x* to *y* labelled *a*. If the distance between points *x* and *y* is at most *b*, then an arc is created from *y* to *x* labelled *b*. If it is known that *x* precedes *y*, but the maximum and minimum distances are unknown, then an arc is created from *x* to *y* labelled 0 and an arc is created from *y* to *x* labelled infinity. The temporal constraint graph for the situation in Figure 7.15(a) is shown in Figure 7.15(b).

7.8.2 The AnalyseLongestPaths algorithm

Given a temporal constraint graph, the AnalyseLongestPaths algorithm checks if temporal constraints are satisfiable, and if so, returns the earliest possible times for each of the time points in the graph:

```
PROCEDURE AnalyseLongestPaths(V, E, length, lower_bound)
/* (V, E) is a directed graph; length(c) returns the length of an arc c ;
   lower_bound(p) returns the earliest starting time of point p in V;
   x[i] stores the updated lower bound for point i in V; */
/* AnalyseLongestPaths labels all x[i] */
BEGIN
   FOR each i in V DO x[i] ← lower_bound(i);
   Converged ← False;
   Counter ← 0;
   WHILE (NOT Converged) DO
      BEGIN
         Converged ← True;
         FOR j = 1 to |Z| DO
            FOR each k such that j→k is in E DO
               IF (x[k] < x[j] + length(j→k)) THEN
                  BEGIN
                     x[k] ← x[j] + length(j→k);
                     Converged ← False;
                  END
         Counter ← Counter + 1;
         IF Counter > |Z| THEN return(NIL);   /* over-constrained */
      END
   /* on exit of the WHILE loop, all the constraints are satisfied */
   return(x);          /* x is the array of all the variables */
END /* of AnalyseLongestPaths */
```

Input to AnalyseLongestPaths is a temporal constraint graph (*V, E*) plus two functions: *length* maps every arc to a numerical value which represents its length; *lower_bound* maps every node to a numerical value which represents its lower bound. If no boundary constraints are specified in the problem, then all the lower bounds may be assigned the value 0.

An array *x* is used to store the value assigned to the time points in the graph. The program initializes each point to the lower bound (i.e. earliest starting time) which is input to the program. Then it updates these lower bounds by propagating the constraints from its preceding nodes. The idea is very similar to the one used in AC-1 in Chapter 4. If any lower bound is updated, then all the constraints in the graph are re-examined. This can easily be improved (following the ideas of AC-2, AC-3 and AC-4) so that constraints are propagated to all *successors* of the updated nodes only. (A successor of a node *x* is a node *y* such that $x \rightarrow y$ is an arc in a directed graph). AnalyseLongestPaths does not insist on the ordering under which the arcs are processed in the inner for loop.

A constraint should never be propagated more than *n* times, where *n* is the number of nodes in the graph. If this happens, it indicates that the value of a node is updated because of its own update. In this case, one can conclude that the constraints are not satisfiable. The Counter helps us to detect such situations. The WHILE loop terminates when no lower bound has been updated.

The AnalyseLongestPaths algorithm finds the longest possible distance from every node to its successor nodes in the graph (hence its name). The AnalyseLongestPaths algorithm finds (or updates) the lower bounds of each node in the graph. It can be modified to the AnalyseShortestPaths algorithm which finds the upper bounds of the nodes.

```
PROCEDURE AnalyseShortestPaths(V, E, length, upper_bound)
/* (V, E) is a directed graph; length(c) returns the length of an arc c;
   upper_bound(p) returns the latest starting time of point p in V;
   y[i] stores the updated upper bound for point i in V; */
/* AnalyseShortestPaths labels all y[i] */
BEGIN
   FOR each i in V DO y[i] ← upper_bound(i);
   Converged ← False;
   Counter ← 0;
   WHILE (NOT Converged) DO
      BEGIN
         Converged ← True;
         FOR j = 1 to |Z| DO
            FOR each k such that k→j is in E DO
```

```
                IF (y[k] > y[j] – length(j→k)) THEN
                    BEGIN
                        y[k] ← y[j] – length(j→k);
                        Converged ← False;
                    END
            Counter ← Counter + 1;
            IF Counter > |Z| THEN return(NIL);   /* over-constrained */
        END
    /* on exit of the WHILE loop, all constraint have been satisfied*/
    return(y);          /* y is the array of all the variables */
END /* of AnalyseShortestPaths*/
```

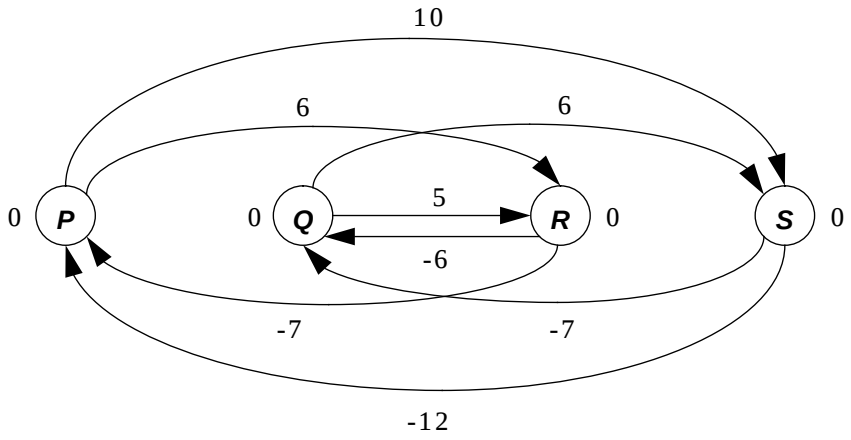
After running both AnalyseLongestPaths and AnalyseShortestPaths on a temporal constraint graph, one may obtain both the lower bounds and the upper bounds for all the time points in the graph. Figure 7.16(a) shows a temporal constraint graph which is unsatisfiable. It is basically a replica of the graph in Figure 7.15(a), except that the bounds of the distances for intervals (*P, R*) and (*Q, S*) are increased. This graph is unsatisfiable because from intervals *A, B* and *C*, one can see that the overlapping part of *B* and *C* is at most (7 + 7) – 10 = 4 units of time. However, the minimum duration of interval *D* is 5, which is greater than 4. If the AnalyseLongestPaths procedure is applied to this graph, it can be found that (*P, S, Q, R, P*) forms a loop, as indicated in Figure 7.16(b). At the situation shown in Figure 7.16(b), the lower bound of *P* could have been increased to 1 (because the lower bound for *R* is 8 at the moment, and the distance from *R* to *P* is -7). Figure 7.17 shows the space searched by AnalyseLongestPaths, assuming that the constraints are propagated in a depth-first manner. It should not be difficult to see that if the temporal constraint graph is satisfiable, the search should never go deeper than the (*n* + 1)-th level, where *n* is the number of nodes in the graph.

One nice property of the above two algorithms is that constraints can be added incrementally. After the upper and lower bounds of the points are computed, new constraints may be added to the constraint graph. Instead of computing the bounds from scratch, these algorithms may start with the values computed in the past so as to save computation.

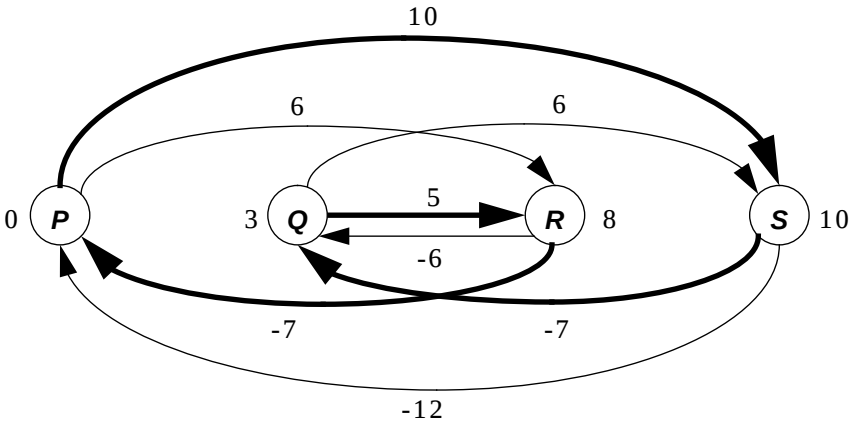
Programs 7.4, *alp.plg*, and 7.5, *asp.plg*, show possible Prolog implementations of the AnalyseLongestPaths and the AnalyseShortestPaths algorithms.

7.9 Summary

In this chapter, we have looked at techniques which, by exploiting the specific features of a CSP, attempt to either reduce the space searched or the complexity in computation.



(a) Example of an input temporal constraint graph to the Analyse-LongestPaths procedure — all lower bounds are initialized to 0



(b) The constraint graph after propagation of the constraints on the highlighted arcs once — a loop is found, hence the constraints are unsatisfiable

Figure 7.16 Example of an unsatisfiable temporal constraint graph detected by the AnalyseLongestPaths procedure

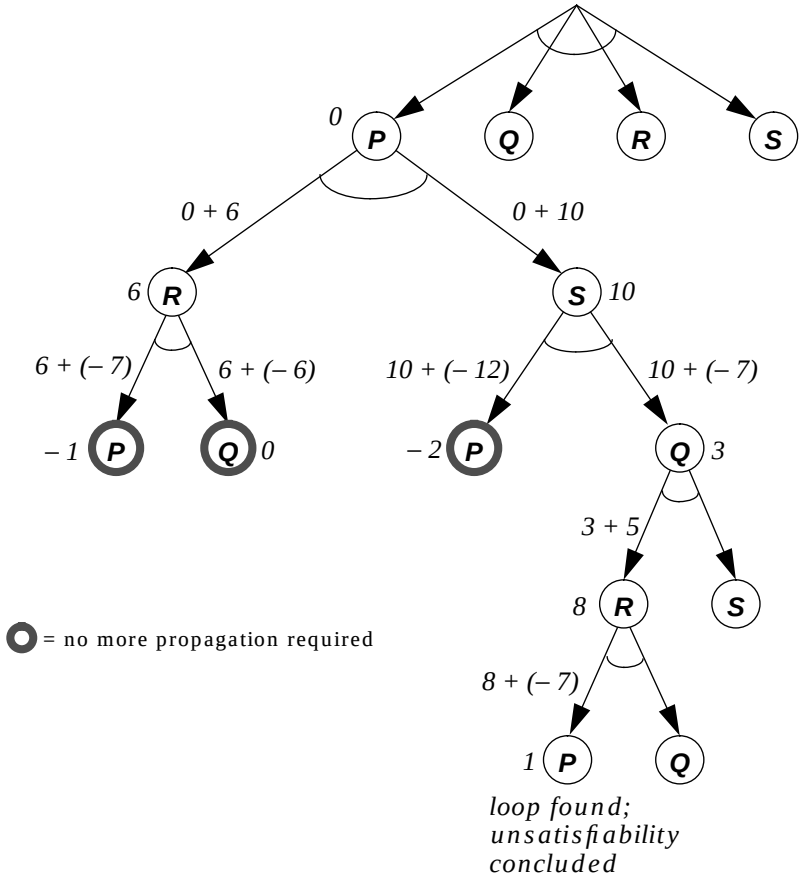


Figure 7.17 Possible space searched by AnalyseLongestPaths for the temporal constraint graph in Figure 7.16

To start with, we have identified some "easy problems" for which efficient algorithms have been developed. If the primal graph of a CSP is not connected, then this problem can be decomposed into independent subproblems which can be solved separately. We have introduced the concept of *k*-trees, and pointed out that if the constraint primal graph of a CSP forms a *k*-tree for some small *k*, then this problem is also easy to solve. A CSP which constraint graph forms a 1-tree (an ordinary tree) can be solved by first reducing it to directional arc-consistent (DAC), and then searching in a backtrack-free manner. If a problem can be recognized as a *k*-tree for

a small k, then by maintaining strong k-consistency in the problem, one needs no backtracking in searching for solutions. When the constraint graph forms a tree, the problem can be solved in $O(na^2)$, where n is the number of variables in the problem, and a is the maximum domain size. When constraint graph forms a k-tree, the problem can be solved in $O(na^{k+1})$ time and $O(na^k)$ space.

Some problems can be reduced to "easy problems" if redundant constraints in them can be identified and removed. One type of redundant constraint, namely path-redundant constraints, and an algorithm for identifying them have been introduced. However, it must be realized that most problems cannot be reduced to "easy problems" through removing redundant constraints.

We have introduced the cycle-cutset method as a dynamic search strategy which identifies the minimal cycle-cutset in an ordering, so that after the variables which form a cutset have been labelled, the Tree_search algorithm can be invoked. The effectiveness of this method very much depends on the size of the cycle-cutset. The overall complexity of the cycle-cutset method is $O(na^{c+2})$, where n is the number of variables in the problem, a is the maximum domain size, and c is the size of the cutset.

The tree-clustering method is a method which attempts to reduce the complexity of a CSP by transforming it into equivalent problems, decomposing it, and then solving the decomposed subproblems. The solutions for the decomposed problems are combined using the Tree_search algorithm. The complexity of the tree-clustering method is $O(ka^{2r})$, (possibly optimized to $O(kra^r \log (a))$), where k is the number of clusters, a is the maximum domain size, and r is the number of variables in the largest cluster in the problem.

We have also summarized the interesting observation that when (i, j)-consistency is maintained in a CSP, then if the nodes in the constraint graph of the CSP are ordered in such a way that it's j-width equals i, the search for solutions under this ordering is j-level backtrack-bounded.

Finally, partly motivated by temporal reasoning, CSPs (under the extended definition which allows infinite domain sizes) with numerical variables and binary linear constraints are studied. The AnalyseLongestPaths and AnalyseShortestPaths algorithms have been introduced, specialized linear programming techniques for finding the lower bounds and upper bounds of the time points.

Figure 7.18 summarizes some sets of special CSPs and the specialized techniques introduced in this chapter for tackling them.

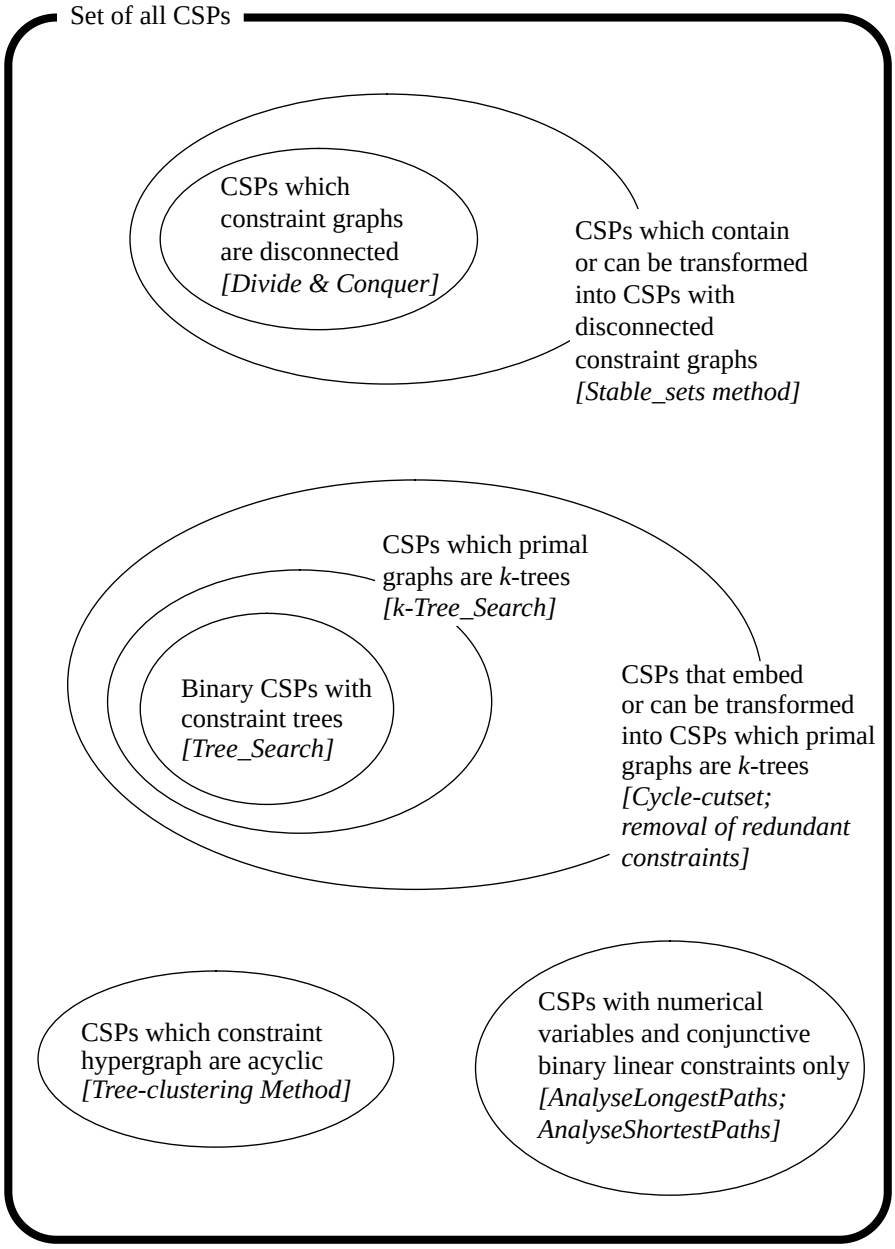


Figure 7.18 Some special CSPs and specialized techniques for tackling them (techniques in italic)

7.10 Bibliographical Remarks

The term "easy problems" was coined by Dechter and Pearl [1988a], though the sufficient condition for backtrack-free search is due to Freuder [1982]. Freuder [1990] further extends this idea to *k*-trees, and presents the W algorithm. Dechter and Pearl [1992] also study the problem of identifying structures in CSPs so as to apply specialized techniques.

The idea of removing redundant constraints to partition the problem into independent sub-problems is introduced by Dechter & Dechter [1987]. Meiri *et al*. [1990] propose removing redundant constraints to reduce the problem to one whose constraint graph is a tree. Their work focuses on binary constraints. A number of preprocessing techniques are compared empirically by Dechter & Meiri [1989]. The cycle-cutset method and the algorithm CCS are introduced by Dechter & Pearl [1987]. The problem of finding cutsets of a minimum size is akin to the *feedback node set problem* described by Carré [1979]. The stable set method is proposed by Freuder & Quinn [1985], who also present the pseudo-tree search algorithm.

The idea of the tree-clustering method comes from database research (e.g. see Beeri *et al*., 1983 and Maier, 1983) and it is brought into CSP research by Dechter & Pearl [1988b, 1989]. The Fill_in-1 algorithm for generating chordal primal graphs is adopted from Tarjan & Yannakakis [1984], who also present an improved algorithm which takes $O(m+n)$ time to complete, where m is the number of arcs and n is the number of nodes in the graph. The general algorithm for identifying maximum cliques can be found in Bron & Kerbosch [1973] and Carré [1979]. The general algorithm for finding join-trees (the Establish_constraints-1 algorithm) is modified from *Graham's Algorithm*, which is used to determine whether a hypergraph is acyclic (see Beeri *et al*., 1983]). The Establish_constraints-2 algorithm is due to Maier [1983]. Gyssens *et al*. [1992] propose an alternative way to decompose problems and attempt to reduce the size of the largest cluster.

Jégou [1990] introduces the *cyclic-clustering method*, which combines the cycle-cutset method and the tree-clustering method. However, it is not difficult to show that the worst case complexity of the cyclic-clustering method is greater than that of the tree-clustering method.

Freuder [1985] establishes the necessary conditions for *b*-bounded search. Dechter *et al*. [1991] formally define the *temporal constraint satisfaction problem* (TCSP). There the class of CSPs that we discuss in Section 7.8 are named *simple temporal problems* (STPs). The AnalyseLongestPaths algorithm is introduced by Bell & Tate [1985] for reasoning with metric time in AI planning. The *Floyd-Warshall algorithm* in Papadimitriou & Steiglitz [1982] uses basically the same principle, but assumes no boundary constraints. Hyvönen [1992] and van Beek [1992] both study algorithms for temporal reasoning.

Apart from the topology of the constraint graph and the variable types, other domain specific characteristics can be exploited. For example, if all the constraints are *monotonic*, *functional* or in general *convex*, the problem can be solved efficiently (see van Hentenryck *et al*., 1992; Deville & van Hentenryck, 1991; and van Beek 1992).