Chapter 9

Tree-Decomposition Methods

As noted in Chapters 4 and 8, topological characterization of tractable classes is centered on the graphical parameter introduced earlier called *induced-width* or *tree-width*. In this chapter, we provide related schemes that are based on the notion of tree-decompositions. These methods are applicable across a wide variety of tasks, not exclusively to constraint processing, and they have received different names in different research areas, such as join-tree clustering, clique-tree clustering and hyper-tree decompositions. We will refer to these schemes by the umbrella name *tree-clustering*. The complexity of these methods is governed by the induced-width, the same parameter that controls the performance of directional (relational) consistency. A related parameter, called hyper-tree width, that is based on the network's hypergraph characterization will also be introduced.

9.1 Acyclic Networks

9.1.1 Solving acyclic problems

In Chapter 4, we showed that trees of binary constraints can be processed efficiently. The primal graph of such binary constraint trees has induced-width of 1. The notion of constraint trees can be extended beyond binary constraints to problems having scope higher than 2, using the notions of hyper-graphs and hyper-trees, leading to the class of *acyclic constraint networks*.

To review, a hypergraph is a structure $\mathcal{H} = (V, S)$ that consists of a set of vertices $V = \{v_1, .., v_n\}$ and a set of subsets of these vertices $S = \{S_1, ..., S_l\}$, $S_i \subseteq V$, called hyperedges. The hyperedges differ from regular edges in that they "connect" more than two variables each.

A hypergraph $\mathcal{H} = (V, S)$ can be mapped to a regular graph called a *dual graph* $\mathcal{H}^{dual}$ (for more details, see Chapter 1). The nodes of the dual graph are the hyperedges from the hypergraph, and a pair of such nodes is connected if they share vertices in V. The arc that connects two such nodes is labeled by the shared vertices. Formally, given $\mathcal{H} = (V, S)$, $\mathcal{H}^{dual} = (S, E)$ where $S = \{S_1, ..., S_l\}$ are edges in $\mathcal{H}$, and $(S_i, S_j) \in E$ iff $S_i \cap S_j \neq \emptyset$. A primal graph of a hypergraph $\mathcal{H} = (V, S)$ has V as its set of nodes, and any two nodes are connected by an arc if they appear in the same hyperedge. Note that if all the constraints of a network $\mathcal{R}$ are binary, then its hypergraph is identical to its primal graph.

Any constraint network $\mathcal{R} = (X, D, C = \{R_1, ..., R_t\})$ can be associated with a hypergraph $\mathcal{H}_{\mathcal{R}} = (X, H)$, where X is the set of nodes (variables), and H is the set of scopes of the constraints in C, namely $H = \{S_1, ..., S_l\}$ where for every i, S_i is the scope of R_i. Therefore, the dual graph of a constraint hypergraph associates a node with each constraint scope and an arc for each two nodes sharing variables. This facilitates the transformation of a non-binary constraint problem into a binary one, called the dual problem, as follows: The variables of the dual problem are the constraints, their domains are the legal tuples of the constraint, and there is a binary constraint between any two dual variables that share original variables, enforcing equality on the values assigned to the shared variables (see Chapter 2). Therefore, if a problem's dual graph happens to be a tree, it means that the dual constraint problem, being a tree, can be solved efficiently by the tree-solving algorithm.

It turns out, however, that sometimes, even when the *dual graph* does not look like a tree, it is, in fact, a tree in the sense that some of its arcs (and their associated constraints) are *redundant* and can be removed, leaving behind a tree structure. A constraint is considered redundant if its removal from the constraint network does not change the set of all solutions. Normally, it is not easy to recognize redundant constraints. However, in the dual representation, some redundancies are easy to identify: since all the constraints in the dual network enforce equalities (over shared variables), a constraint (and its corresponding arc) can be deleted if the variables labeling the arc are shared by every arc along an *alternate* path between the two end points. This is because the alternate path (of constraints) enforces that equality. Removing such constraints does not alter the problem.

Example 9.1.1 Figure 9.1 depicts the *hypergraph* (a), the *primal* (b) and the *dual graph* (c) representations of a constraint network with variables A, B, C, D, E, F and with constraints on the scopes (ABC),(AEF), (CDE), and (ACE). The specific constraints are irrelevant to the current discussion. They can be arbitrary relations over domains of $\{0, 1\}$ such as $C = A \vee B$, $F = A \vee E$ and so on. We see that the arc between (AEF) and (ABC) in Figure 9.1(c) is redundant because variable A appears also along the alternative path $(ABC)-AC-(ACE)-AE-(AEF)$, and so a consistent assignment to A is ensured by these constraints even if the constraint between AEF and ABC is removed. Likewise, the arcs labeled E and C are also redundant, and their removal yields the graph in 9.1(d).
$\square$

The property that ensures such legitimate arc removal is known as the *running intersection property (r.i.p)* or *connectedness* property. The running intersection property (r.i.p.) can be defined over hypergraphs or over their dual graphs, and it is used to characterize equivalent concepts such as *join-trees* (defined over dual graphs) or *hypertrees* (defined over hypergraphs).

Definition 9.1.2 (connectedness, join-trees, hypertrees, and acyclic networks) *Given a dual graph of a hypergraph, an arc subgraph of the dual graph satisfies the connectedness property iff for each two nodes that share a variable, there is at least one path of labeled arcs, each containing the shared variables. An arc subgraph of the dual graph that satisfies the connectedness property is called a* join-graph*. A join-graph that is a tree is called a* join-tree*. A hypergraph whose dual-graph has a join-tree is called a* hypertree*. A constraint network whose hypergrpah is a hyper-tree is called an acyclic network.*

Example 9.1.3 Considering again the graphs in Figure 9.1, we can see that the running intersection property is maintained even if the arc between AEF and ABC is removed. It is possible to remove also the arcs labeled C and E, while not violating connectedness, thus turning the join-graph in Figure 9.1(c) into a *join-tree* (Figure 9.1(d)). The hypergraph in Figure 9.1(a) has a join-tree and is therefore a hypertree. □

An acyclic constraint network can be solved efficiently as follows: Because the constraint problem has a join-tree, its dual problem is a tree of binary constraints and can therefore be solved efficiently by the tree-solving algorithm. In Figure 9.2, we reformulate the tree algorithm for solving acyclic problems. The algorithm assumes that domain constraints are already *absorbed* into the relevant relations, namely, any tuple in a relation that has an illegal domain value of some variable, is removed.

Example 9.1.4 Consider the tree dual problem in Figure 9.1(d), and let us assume that the constraints are given by: $R_{ABC} = R_{AEF} = R_{CDE} = \{(0,0,1)(0,1,0)(1,0,0)\}$ and $R_{ACE} = \{(1,1,0)(0,1,1)(1,0,1)\}$. Assume the ordering $d = (R_{ACE}, R_{ABC}, R_{AEF}, R_{CDE})$. When processing R_{CDE}, its parent relation is R_{ACE}; we therefore generate $\pi_{ACE}(R_{ACE} \bowtie R_{ABC})$, yielding the revised relation $R_{ACE} = \{(0,1,1)(1,0,1)\}$. Next, processing R_{AEF} (likewise connected to R_{ACE}), we generate relation $R_{ACE} = \pi_{ACE}(R_{ACE} \bowtie R_{AEF}) = \{(0,1,1)\}$. Note that the revised relation R_{ACE} is now being processed. Subsequently, processing R_{CDE}, we generate: $R_{ACE} = \pi_{ACE}(R_{ACE} \bowtie R_{CDE}) = \{(0,1,1)\}$. Then a solution can be generated by picking the only allowed tuple for R_{ACE}, $A = 0, C = 1, E = 1$, extending it with a value for D that satisfies R_{CDE} which is only $D = 0$, and similarly, extending the assignment to $F = 0$ and $B = 0$ to satisfy R_{AEF} and R_{ABC}. □

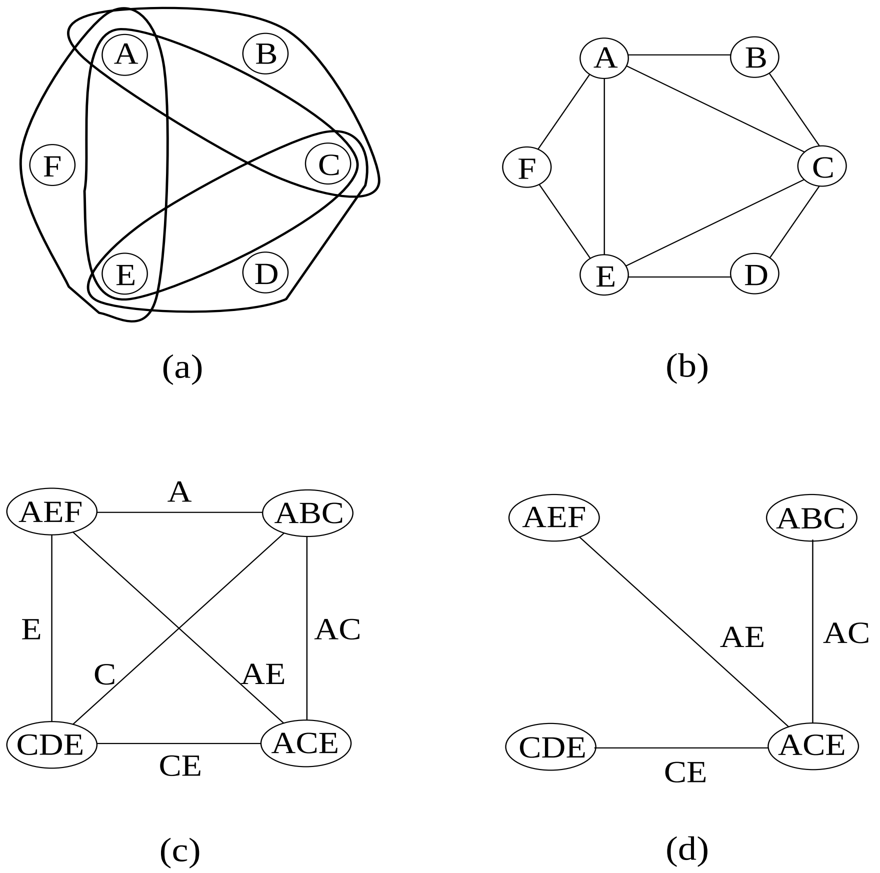

Figure 9.1: (a)Hyper, (b)Primal, (c)Dual and (d)Join-tree constraint graphs of a CSP.

Algorithm acyclic-solving is correct, essentially applying directional arc-consistency from leaves to root (step 2) along the join-tree of the dual problem. Since the complexity of a tree-solving algorithm is $O(nk^2)$ where n is the number of variables and k bounds the size of the domains size, the implied complexity here is $O(r \cdot l^2)$ if there are r constraints, each allowing at most l tuples. However, the complexity can be improved for this special case. The join operation can be performed in time linear in the maximum number of tuples of each relation: Project R_j on the variables shared by R_j and its parent constraint, R_k, an $O(l)$ operation, and then prune any tuple in R_k that has no match in that projection. If tuples are ordered lexicographically, which requires $O(l \cdot logl)$ steps, the join operator has a complexity of $O(l)$, yielding an overall complexity of $O(r \cdot l \cdot logl)$ steps. In summary,

Theorem 9.1.5 (correctness and complexity) *Algorithm acyclic-solving decides the consistency of an acyclic constraint network, and its complexity is $O(r \cdot l \cdot logl)$ steps,*

ALGORITHM ACYCLIC-SOLVING

Input: an acyclic constraint network $\mathcal{R} = (X, D, C)$ $C = \{R_1, ..., R_t\}$.
S_i is the scope of R_i. A join-tree T of $\mathcal{R}$.
Output: Determine consistency, and generate a solution.
1. $d = (R_1, ..., R_t)$ is an ordering such that every relation
   appears before its descendent relations in the tree rooted at R_1.
2. **for** $j = t$ to 1, for edge (j,k) ,$k < j$, in the tree do
   $R_k \leftarrow \pi_{S_k}(R_k \bowtie R_j)$
   if the empty relation is created, exit, the problem has no solution.
   endfor
3. **return:** The problem is consistent. Generate a solution:
   Select a tuple in R_1. After instantiating $R_1, ..., R_{i-1}$ select a tuple in R_i
   that is consistent with all previous assignments.

Figure 9.2: Acyclic-solving algorithm

where r is the number of constraints and l bounds the number of tuples in each constraint relation.$\Box$

Another distributed version of algorithm ACYCLIC-SOLVING is where every node send the message in step 2 to each one of its neighbors in the join-tree. Once they all recieve this message, each can compute the *minimal relations* over their scopes. We will revisit this issue is subsequent sections.

9.1.2 Recognizing Acyclic Networks

While binary trees can be characterized as having induced-width of 1, for non-binary acyclic networks, the induced-width can be arbitrarily large, because it is at least as large as the number of variables in a hyperedge. Therefore, another parameter that can capture the tractability of acyclic and near-acyclic constraint networks is needed. An extension of induced-width to *hyper-width* provides the desired characterization. In particular, it will be shown that a hypergraph has a hyper-width of 1 iff it is a hypertree.

Several efficient procedures for identifying acyclic networks and for finding a representative *join-tree* were developed in the area of relational databases [116]. Two alternative methods will be discussed. One is based on processing the dual constraint graph and the other on consulting its primal graph.

ALGORITHM DUAL-ACYCLICITY($\mathcal{R}$)
Input: A hypergraph $\mathcal{H}_{\mathcal{R}} = (X, S)$ of a constraint network $\mathcal{R} = (X, D, C)$, $C = \{R_1, ..., R_t\}$ and S_i is the scope of R_i.
Output: A join-tree $T = (S, E)$ of $\mathcal{H}_{\mathcal{R}}$, if $\mathcal{R}$ is acyclic.
1. $T = (S, E) \leftarrow$ Generate a maximum spanning tree of the weighted dual constraint graph of $\mathcal{R}$.
2. **for** every two nodes, u and v in T, do
 if the unique path connecting them in T does not satisfy the connectedness property, exit ($\mathcal{R}$ is not acyclic).
 endfor
3. **return:** R is acyclic, and T is a join-tree of $\mathcal{H}_{\mathcal{R}}$.

Figure 9.3: Algorithm Dual-acyclicity

Dual-based recognition

It turns out [116] that if a hypergraph has a join-tree, the join-tree is a maximum spanning tree of its dual graph when the arcs of the dual graph are weighted by the number of shared variables. Therefore, testing if a hypergraph has a join-tree can be done by generating a maximum spanning tree over its dual graph. Once a spanning tree is obtained, testing connectedness can be accomplished efficiently because there is only one path between every pair of nodes in the candidate join-tree. Algorithm *dual-acyclicity*, which accomplishes this, is described in Figure 9.3.

Theorem 9.1.6 (Correctness and complexity) *Algorithm* dual-acyclicity *decides if a constraint network $\mathcal{R}$ is acyclic. If it is, the algorithm returns a join-tree of $\mathcal{R}$'s dual graph. The algorithm's complexity is $O(e^3)$, where e is the number of constraints.*

Proof: The number of nodes in the dual graph equals the number of constraints, e. Since generating a maximum spanning tree of a graph can be done in $O(n^2)$ when n is the number of nodes in the graph, generating the spanning tree here is $O(e^2)$. A brute force approach for testing the connectedness property takes $O(e^3)$, because there are $O(e^2)$ pairs of nodes and testing this property for the unique path between each pair is $O(e)$. For correctness see [116] and exercises. □

Example 9.1.7 Had we applied a maximum spanning tree algorithm to the dual graph in Figure 9.1(c), the three arcs labeled by singleton variables would be removed, resulting in the join-tree in Figure 9.1(d), which, as already observed, satisfies connectedness. □

Primal-based recognition

The second acyclicity recognition scheme is based on the fact that a primal graph of a hypertree must be chordal [12]. For a general hypergraph, the primal graph is a collection of maximal cliques, each corresponding to a hyperedge. Therefore, an alternative approach to recognizing acyclicity is based on testing if the primal graph of the constraint network is chordal. If it is, its maximal cliques should be identified, which is easy to do on chordal graphs, as we noted in Chapter 4. However, since different hypergraphs can map to the same primal graph, there is a need to ensure that the maximal cliques of the primal graph, if it is chordal, coincide with the scopes of the original constraints. This latter property is called *conformality.*

Definition 9.1.8 (conformality) *A chordal primal graph is conformal relative to a constraint hypergraph iff there is one-to-one mapping between maximal cliques and scopes of constraints.*

As discussed in Section 1 of Chapter 4, a graph is chordal if, in a *maximal cardinality ordering*, each node's parents are connected. Therefore, using such an ordering, the chordality of the constraint primal graph can be determined. Subsequently, if the primal graph is chordal, its maximal cliques can be identified in linear time, listing every node and its parents in the ordering. If the maximal cliques coincide with the scopes of the constraints, both conditions for acyclicity (chordality and conformality) are satisfied, and we know that the constraint network is acyclic. This leads to algorithm *primal-acyclicity* described in Figure 9.4.

Theorem 9.1.9 *[116] A hypergraph has a join-tree iff its primal graph is chordal and conformal.*

Example 9.1.10 Consider again the constraint network of Example 9.1.1, whose primal graph is given again in Figure 9.5(a). A max-cardinality ordering is given in 9.5(b). The graph is chordal, since all the parents of each node are connected. The maximal cliques, enumerated from top to bottom are: FAE (the clique of F), DEC (the clique of D), EAC (the clique of E), and ABC (the clique of A). The cliques of B and C are not maximal. Therefore, there are four maximal cliques, and these are identical to the constraints' scopes. The graph is therefore recognized as acyclic. Next, clique FAE is connected to clique EAC, DEC is connected to EAC which is connected to ABC. Now we have a join-tree representation equivalent to the original problem. (See Figure 9.5(c)). $\square$

Primal-acyclicity($\mathcal{R}$)

Input: A network $\mathcal{R} = (X, D, C)$ and its primal graph G
Output: A join-tree iff the problem is acyclic.
1. **Test chordality:** Using a max-cardinality ordering $d = (x_1, ..., x_n)$ of G,
   for i=n to 1 do:
   if the parents of x_i are not connected, exit. The problem is not acyclic.
2. **Test conformality:** Let $C_1, ..., C_t$ be the cliques (a node and all its parents) indexed by their highest variable.
   If the maximal cliques correspond to scopes of constraints, $\mathcal{R}$ is acyclic.
   else, the network is not acyclic. exit.
3. *Create a join-tree:*
   Let $C_1, ..., C_t$ be the cliques indexed by their highest variable.
   Connect every clique to an earlier clique with which it shares a maximal number of variables.
5. **return** the tree of cliques.

Figure 9.4: Algorithm Primal-acyclicity

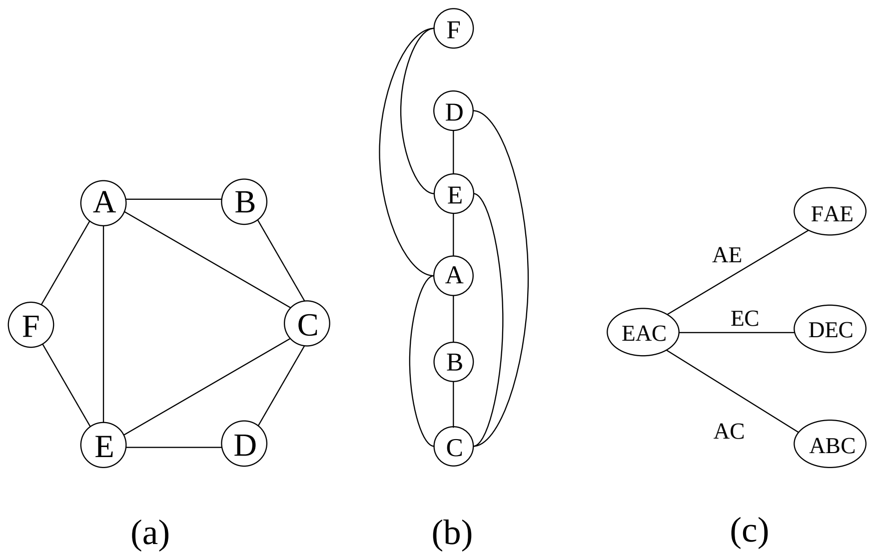

Figure 9.5: (a) A primal graph, (b) a max cardinality ordering, and (c) its join-tree

9.2 Join-tree clustering

9.2.1 Join-tree clustering

Since acyclic constraint networks can be solved efficiently, it is natural to aim at compiling an arbitrary constraint network into an acyclic one. This can be achieved by grouping subsets of constraints into clusters, or subproblems, whose scopes constitute a hypertree, thus transforming a constraint hypergraph into a constraint hypertree. Replacing each subproblem with its set of solutions yields an acyclic constraint problem. If the trasnformation process is tractable, including both graph transformation and finding all solutions to each subproblem, the resulting algorithm is polynomial. This compilation process is called *join-tree clustering*.

The graphical input to the scheme is a constraint hypergraph $\mathcal{H} = (X, H)$, where H is the set of scopes of the constraint network, and the output is a hypertree $\mathcal{S} = (X, S)$ and a partition of the original hyperedges into the new hypertree edges.

Definition 9.2.1 (hypertree embedding of a hypergraph) *A* hypertree *embedding of a hypergraph $\mathcal{H} = (X, H)$ is a hypertree $\mathcal{S} = (X, S)$ s.t., for every $h \in H$ there is $h_1 \in S$ s.t. $h \subseteq h_1$.*

Once a hypertree $\mathcal{S}$ is constructed, each of its edges defines a subproblem that contains a constraint if its scope is contained in the hyperedge. Each subproblem, is then solved, and its set of solutions is a new constraint whose scope is the hyperedge. The resulting problem has one constraint per hyperedge of $H\prime$, and by construction, it is acyclic.

There are various specific methods that decompose a hypergraph into a hypertree. The aim is to generate hypertrees that have small-sized hyperedges because this implies small constraint subproblems. The most popular approach manipulates the constraint primal graph. It emerges from the primal recognition process (Figure 9.4). Rather than stopping at Step 1 of algorithm primal-acyclicity (because some parents are not connected), chordality can be enforced by connecting all parents. Subsequently, if conformality is violated (Step 2), it can be also enforced by generating a unique constraint for every maximal clique. The maximal cliques of the resulting chordal graph are the candidate scopes in the formed acyclic network. These cliques define the subproblems that must be independently solved. The procedure, called join-tree clustering (JTC) is described in Figure 9.6.

While any variable ordering can be used in Step 1 of the algorithm, the max-cardinality ordering may be a good choice because when the graph is chordal, no edges will be added in Step 2. The first three steps of the algorithm manipulate only the primal graph, embedding it in a chordal graph (whose maximal cliques make a hypertree), and then identifying its join-tree. Step 4 partitions the constraints into the cliques (the hypertree

JOIN-TREE CLUSTERING (JTC)

Input: A constraint problem $\mathcal{R} = (X, D, C)$ and its primal graph $G = (X, E)$.
Output: An equivalent acyclic constraint problem and its join-tree: $\mathcal{T} = (X, D, C')$
1. Select a variable ordering, $d = (x_1, ..., x_n)$.
2. **triangulation** (create the induced graph along d and call it G^*):
   (**for** $j = n$ to 1 by -1 do $E \leftarrow E \cup \{(i, k) | (i, j) \in E, (k, j) \in E\}$)
3. **Create a join-tree of the induced graph** G^*:
   a. Identify all maximal cliques in the chordal graph (each variable and parents is a clique).
      Let $C_{l_1}, ..., C_{l_t}$ be all such cliques, indexed by their latest variable in the ordering.
   b. Create a tree-structure T over the cliques:
      Connect each C_{l_i} to a C_{l_j} $(j < i)$ with whom it shares largest subset of variables.
4. Place each constraint in one clique containing its scope, and let
   P_i be the constraint subproblem associated with Cl_i.
5. Solve P_i and let R'_i be its set of solutions.
6. Return $C' = \{R'_1, ..., R'_t\}$, the new set of constraints and their join-tree, T.

Figure 9.6: Join-tree clustering

edges); Step 5 solves each clique's subproblem, and thus creates one new constraint for each clique. We conclude,

Theorem 9.2.2 *Algorithm join-tree clustering transforms a constraint network into an equivalent acyclic network.* $\Box$

Example 9.2.3 Consider the graph in Figure 9.7(a), and assume it is a primal graph of a binary constraint network. In this case, the primal and hypergraph are the same. Consider the ordering $d_1 = (F, E, D, C, B, A)$ in Figure 9.7(b). Performing tree-clustering connects parents recursively from the last variable to the first, creating the induced-ordered graph by adding the new (broken) edges of Figure 9.7(b). The maximal cliques of this induced graph are: $Q_1 = \{A, B, C, E\}$, $Q_2 = \{B, C, D, E\}$, and $Q_3 = \{D, E, F\}$. Alternatively, if ordering d_2 in 9.7(c) is used, the induced graph generated has only one added edge. The cliques in this case are: $Q_1 = \{D, F\}$, $Q_2 = \{A, B, E\}$, $Q_3 = \{B, C, D\}$ and $Q_4 = \{A, B, C\}$. The corresponding join-trees of both orderings are depicted in Figure 9.8 (broken arcs are not part of the join-trees). Next, we partition the constraints into the tree-nodes. We place the following subproblems into the nodes: $P_1 = \{R_{FD}\}$ is placed in node (FD), $P_2 = \{R_{BD}, R_{CD}\}$ is placed in node (BCD), $P_3 = \{R_{AB}, R_{AC}\}$ is placed in node (ABC), and $P_4 = \{R_{AB}, R_{BE}, R_{AE}\}$ is placed in (ABE). Next, applying

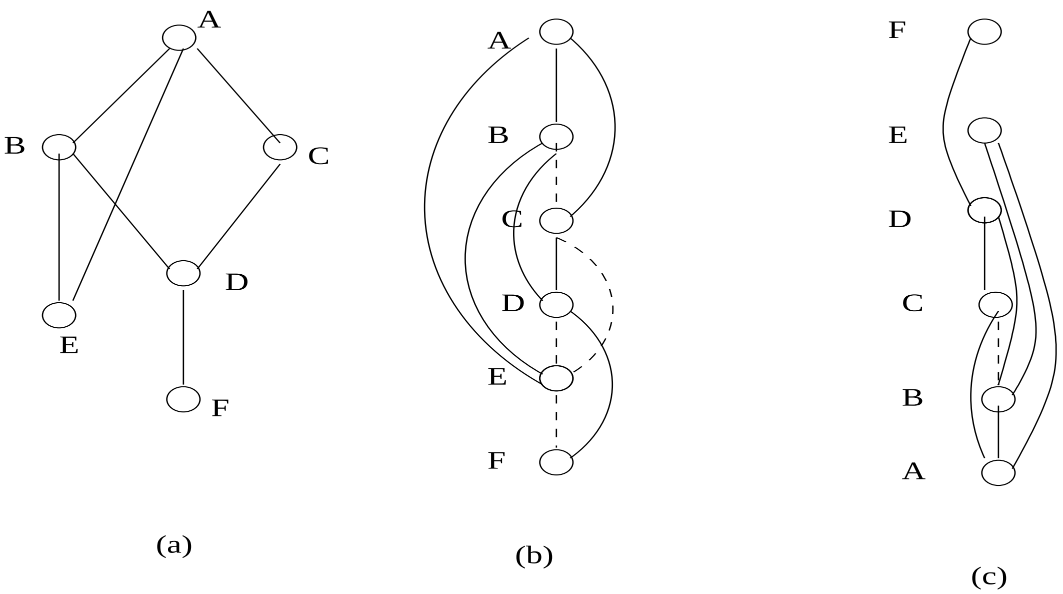

Figure 9.7: A graph (a) and two of its induced graphs (b) and (c). All arcs included.

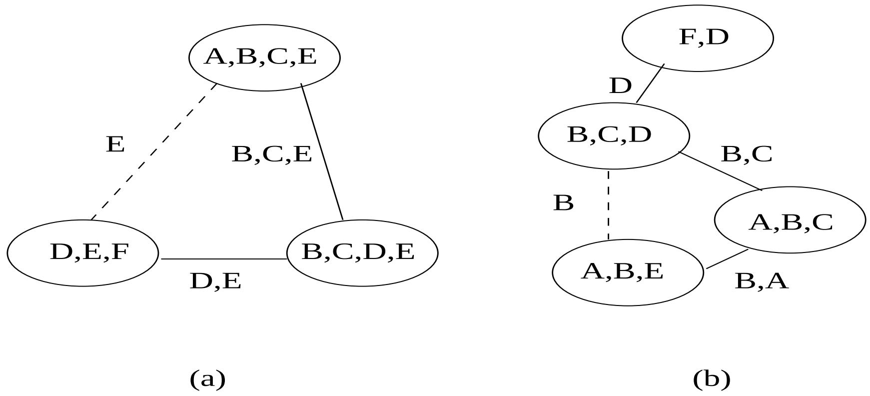

Figure 9.8: Join-graphs of the induced graphs from (a)Figure 9.7(b) and Figure 9.7(c). (All arcs included.) The corresponding join-trees are the same figures with the broken arcs removed.

Step 4 of the algorithm, we solve the subproblems $P_1, ...P_4$, and replace each with its solution, R_1, R_2, R_3, R_4 where R_i is the solution relation of P_i, yielding a desired acyclic network. $\Box$

In general, the maximal clique sizes of the induced ordered graphs equal their induced width+1. The running time of join-tree clustering is dominated by computing the set of solutions of each subproblem. This is determined by the maximal number of variables in a subproblem and is captured by the induced-width. It is easy to see that,

Theorem 9.2.4 (complexity of JTC) *Given a constraint network having n variables and r constraints, the time and space complexity of join-tree clustering is $O(r \cdot k^{w^*(d)+1})$, where r is the number of constraints , k is the maximum size domain, and $w^*(d)$ is the induced width of the ordered graph.*

Proof: Finding a tree-decomposition of a hypergraph (Step 1) is performed over the constraint primal graph requiring $O(n^2)$ steps. The most expensive step is Step 3, computing all the solutions of each subproblem. Since the size of each subproblem corresponds to a clique in the induced (triangulated) ordered graph, it is bounded by the induced width plus one. Solving a problem P_i with at most $w^*(d)+1$ variables having at most r_i constraints is $O(r_i \cdot exp(w^*(d)+1))$. Summing over all subproblems $\sum_i r_i exp(w^*(d)+1)$, yields the desired bound.
$\Box$

9.2.2 Unifying Tree-Decomposition Schemes

Next we describe *JTC* within a more general and formal setting that unifies a variety of such algorithms ippearing n a variety of reasoning areas. The exposition divides tree-clustering algorithms into two parts. The first protion defines the notion of hypergraph embeddings precisely as a *tree-decomposition* without commiting to any specific algorithm that create such a decomposition. The second,is an algorithm that processes a given tree-decomposition (essentialling combining the solving of subproblems acyclic problem solving). The algorithm is described in a distributed fashion as a message passing between adjacent nodes in the tree-decomposition.

Definition 9.2.5 (tree-decomposition) *Let $\mathcal{R} = (X, D, C)$ be a CSP problem. A* tree-decomposition *for $\mathcal{R}$ is a triple $< T, \chi, \psi >$, where $T = (V, E)$ is a tree, and χ and ψ are labeling functions which associate each vertex $v \in V$ with two sets, $\chi(v) \subseteq X$ and $\psi(v) \subseteq C$ that satisfy the following conditions:*

1. *For each constraint $R_i \in C$, there is* at least *one vertex $v \in V$ such that $R_i \in \psi(v)$, and $scope(R_i) \subseteq \chi(v)$.*

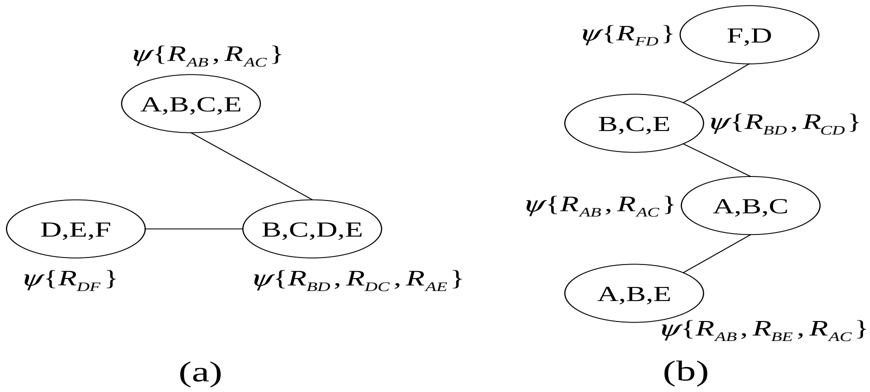

Figure 9.9: Two tree-decompositions

2. *For each variable $x \in X$, the set $\{v \in V | x \in \chi(v)\}$ induces a connected subtree of T. This is the connectedness property.*

Definition 9.2.6 (tree-width, hyper-width, separator) *The tree-width of a tree-decomposition $< T, \chi, \psi >$ is $tw = \max_{v \in V} |\chi(v)|$, and its hyper-width is $hw = \max_{v \in V} |\psi(v)|$. Given two adjacent vertices u and v of a tree-decomposition, a separator of u and v is defined as $sep(u, v) = \chi(u) \cap \chi(v)$.*

Example 9.2.7 Consider the binary constraint problem whose primal graph appears in Figure 9.7(a). The join-trees in Figure 9.8(a) and (b) can be described again in Figure 9.9, using the two labeling functions. □

Graphically, a tree-decomposition defines a hypertree embeddings of a hypergraph. The smallest tree-width and hyper-width among all such embeddings are called the *tree-width* and the *hyper-width* of the constraint hyper graph, respectively.

Once a tree-decomposition is available, algorithm *Cluster-Tree Elimination (CTE)* in Figure 9.10, processes the decomposition. The algorithm is applicable to a wide variety of tasks where the join and project operators are given a general interpretation.

The algorithm is presented as a message-passing algorithm, where each vertex of the tree sends a constraint to each of its neighbors. If the tree contains m edges, then a total of $2m$ messages will be sent. Node u takes all the constraints in $\Psi(u)$ and all the constraint messages received by u from all adjacent nodes, joins them, and projects the combined function onto the separator of u and v. The projected constraint is then sent to v (remember that v is adjacent to u in the tree). For clarity of exposition we do not mention specail case processing of instantiated constraints that share no variables with the eliminated variables. Those will just be passed along sepearately in the message.

Cluster tree-elimination (CTE)
Input: A tree decomposition $< T, \chi, \psi >$ for a problem $\mathcal{R} =< X, D, C >$.
Output: An augmented tree whose nodes are clusters containing the original constraints as well as messages received from neighbors. A decomposable problem for each node v.
Compute messages:
For every edge (u, v) in the tree, do

- Assume that node u has received messages from all adjacent vertices, except maybe from v and let $m_{(u,v)}$ denote the message sent by vertex u to vertex v.
  Let, $cluster(u) = \psi(u) \cup \{m_{(i,u)} | (i, u) \in T,\ i \neq v\}$
  Compute and send to v:

$$m_{(u,v)} = \pi_{sep(u,v)}(\bowtie_{R_i \in cluster(u)} R_i) \tag{9.1}$$

Return: A tree-decomposition augmented with constraint messages. For every node $u \in T$, return the decomposable subproblem $cluster(u) = \psi(u) \cup \{m_{(i,u)} | (i, u) \in T\}$

Figure 9.10: Algorithm cluster-tree elimination (CTE)

The output of the algorithm is the original tree-decomposition where each node is augmented with the constraints sent from neighboring nodes called clusters. For each node the augmented set of constraints constitue a *minimal subproblem* relative to the input constraint problem $\mathcal{R}$. Intuitively, a subproblem of a constraint network is minimal if you can answer any query on it correctly without having to refer back to information in the whole network. More precisely, a subproblem over a subset of variables Y is minimal relative to the whole network, if its set of solutions is identical to the projection of the networks' solutions on Y.

Definition 9.2.8 (minimal subproblem) *Given a constraint problem $\mathcal{R} = (X, D, C)$ and a subset of variables $Y \subseteq X$, a subproblem over Y, $\mathcal{R}_Y = (Y, D_Y, C_Y)$, is decomposable relative to $\mathcal{R}$ iff $sol(R_Y) = \pi_Y sol(\mathcal{R})$ where $sol(\mathcal{R})$ is the set of all solutions of network $\mathcal{R}$.*

Node activation can be asynchronous. Convergence is guaranteed, but it may take as long as the diameter of the tree in the worst case. If processing is performed from leaves to root and back, convergence is guaranteed after two passes, where only one constraint message is sent on each edge in each direction. The algorithm in Figure 9.10 is described in a synchronized form.

Since CTE can be shown to be equivalent to generating an acyclic constraint problem and its subsequent solving by a tree-solving algorithm we can conclude,

Theorem 9.2.9 (correctness and completeness) *Given a tree-decomposition of a constraint problem $\mathcal{R}$, algorithm CTE is sound and complete for generating minimal subproblems over $\chi(v)$ for every node v in the tree.* □

Complexity of CTE

Algorithm CTE can have several variations which will influence its time and space complexities in a subtle way. As described in Figure 9.10, its time complexity can be far larger than its space complexity. At a first glance, it seems that the space complexity is also exponential in w^*. Indeed, if we first record the joined relation in Equation 9.1 and subsequently project on the separator, we will have space complexity exponential in w^*. However, we can interleave the join and project operations and restrict the space complexity to be identical to the size of the constraint message that is sent. In Equation 9.1, we compute the message m which is a function defined over the separator, sep, because all the variables in the *eliminator*, $elim(u) = \chi(u) - sep$ are eliminated by projection. This can be implemented by enumeration (or search) as follows: For each assignment v to $\chi(u)$, we can test if v is consistent with each constraint in cluster(u), and if it is, we will project the tuple v over sep, creating v_{sep}, and add it to the relation $m(sep)$.

Theorem 9.2.10 (Time-space complexity by tree-width) *Let N be the number of nodes in a tree decomposition, w^* be its tree-width, sep be its maximum separator size, r be the number of constraints, hw is its hyper-width and deg is the maximum degree in T. The time complexity of CTE is $O((r+N) \cdot deg \cdot exp(w^*))$ and the space complexity is $O(N \cdot exp(sep))$.*

Proof. Let deg_u be the degree of u. The time complexity of processing a node u is $deg_u \cdot (|\psi(u)| + deg_u - 1) \cdot exp(|\chi(u)|)$, because the number of constraints that are processed by u for each each neighbor is $\psi(u) + deg_u - 1$. By bounding deg_u by deg and $\chi(u)$ by w^*, and summing over all nodes, we can bound the entire time complexity by $O(deg \cdot (r+N) \cdot exp(w^*))$. Assuming the enumeration algrithm described above, the time is $exp(w^*)$ while the space is $exp(sep)$. For each edge, CTE will record two constraints. Since the number of edges is bounded by N, and the size of each constraint we record is bounded by $exp(sep)$, the space complexity is bounded by $O(N \cdot exp(sep))$. If $r \geq n$, this yields complexity of $O(deg \cdot r \cdot exp(w))$. It is possible to have an implementation of the algorithm whose time complexity will not depend on deg. But this improvement will be more expensive in memory [155, 89]. □

The performance of CTE can also be bounded exponentially in the hyper-width as well, but this may also require paying more in terms of space complexity. Assume that every constraint has at most t tuples. Clearly, a join operation requires at most t^2 time and space. Since we have at most hw relations in a cluster. we can first join them and record the resulting relation, using t^{hw} time and space. Joining the resulting relation and the message from each neighbor (whose size is also $O(t^{hw})$), may take at most $O(t^{2hw})$ per neighbor, yielding $deg \cdot t^{2hw}$ per outgoing message. Since this only tightens the relation generated at node u, and since the message sent to each neighbor is a projection on a separator of this tightened relation, it yields $2deg \cdot t^{2hw}$ per node and overall $O(2N \cdot t^{2hw})$ time complexity and $O(N \cdot t^{hw})$ space complexity. It is important to note that t can be exponential in the tree-width (or induced-width) of the problem. However, when the input relations are tight and expressed explicitly, the hyper-width can provide a tighter bound on complexity then that provided by the tree-width, and can characterize a wider class of tractability.

Theorem 9.2.11 (Time-space complexity by hyper-width) *Let N be the number of nodes in a tree decomposition, T hw be its hyper-width, r be the number of constraints, hw be its hyper-width, deg be the maximum degree in T. The time complexity of CTE is $O(N \cdot exp(2hw))$ and the space complexity is $O(N \cdot exp(hw))$.*

Join-tree clustering as tree-decomposition

Algorithm JTC is committed to a specific algorithm for creating the tree-decomposition (Steps 1-3 of JTC). As mentioned, solving the subproblems in the tree and then applying the tree-solving algorithm is equivalent to CTE. JTC is a compilation algorithm. Because it generates the full set of solutions for each node, its space complexity is exponential in the tree-width w^*, while, as we saw, CTE's space complexity is exponential in the separator's size only. On the other hand, while the time complexity of CTE is $O(r \cdot deg \cdot exp(w^*))$ if $N \leq r$, the time complexity of JTC is just $O(r \cdot exp(w^*))$. This is because the relation associated with a node is recomputed for each neighboring node by CTE in order to save space. Clearly, this distinction matters only if there is a substantial difference between the tree-width and the maximum separator size of a given tree-decomposition. The following example demonstrates that there are tree-decompositions that will not be created by the first 3 steps of JTC.

Example 9.2.12 Consider a problem having constraints defined on all pairs of variables whose graph is complete. Clearly, the only possible tree-decomposition created by triangulation will have one node containing all the variables and all the constraints. An alternative tree-decomposition has node C_1 whose variables are $\{1, ..., n\}$ and whose constraints are defined over the pairs of variables:

$$\{(1,2)(3,4),....(i,i+1)(i+2,i+3)....\}.$$

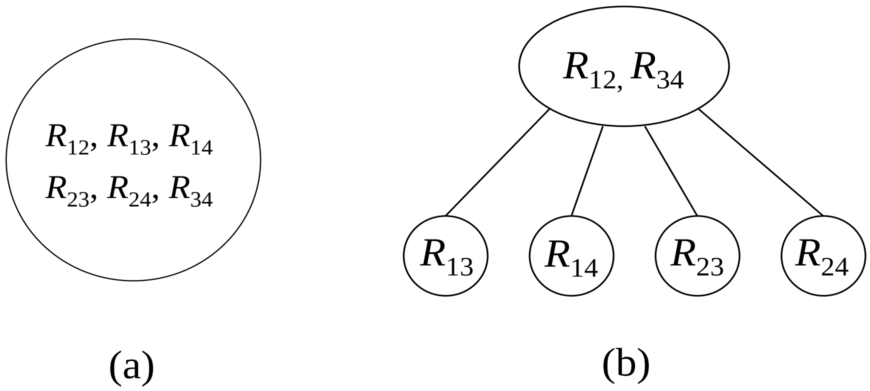

Figure 9.11: Two tree-decompositions of a complete binary constraint network (a) a join-tree (b) a non-join-tree

Then there is a node, $C_{i,j}$ for each other constraint that is not included in C_1, and the tree connects C_1 with each other node. Figure 9.11 demonstrates these two cases for a network having four nodes $\{1, 2, 3, 4\}$ and binary constraints $\{R_{12}, R_{13}, R_{14}, R_{23}, R_{24}, R_{34}\}$. □

9.3 Adaptive-consistency as tree-decomposition

This section shows that the bucket-elimination algorithm, adaptive-consistency (AC), discussed in Chapters 4 and 8, can be viewed as a message-passing algorithm along a bucket-tree which is a special case of tree-decomposition. We first redefine the notion of buckets.

Let $\mathcal{R} = (X, D, C)$ be a problem and d an ordering of its variables, $d = (x_1, ..., x_n)$. Let $B_{x_1}, ..., B_{x_n}$ be a set of buckets, one for each variable. Each bucket B_{x_i} contains those constraints in C whose latest variable in d is x_i. A *bucket-tree* of $\mathcal{R}$ and an ordering d has buckets as its nodes, and bucket B_x is connected to bucket B_y if the constraint generated in bucket B_x by adaptive-consistency is placed in B_y. The variables of B_{x_i} are those appearing in the scopes of any of its original constraints as well as those received from other buckets. Therefore, in a bucket tree, every node B_x has one parent node B_y and possibly several child nodes $B_{z_1}, ...B_{z_t}$. The structure of the bucket-tree can also be extracted from the induced-ordered graph of $\mathcal{R}$ along d using the following definition.

Definition 9.3.1 (bucket tree, graph-based) *Let (G, d) be the induced graph along d of a problem $\mathcal{R}$ whose primal graph is G. Each variable x and its earlier neighbors in the induced-graph is a variable in bucket B_x. The nodes of the bucket-tree are the n buckets. Each node B_x points to B_y (or, B_y is the parent of B_x) if y is the latest earlier neighbor*

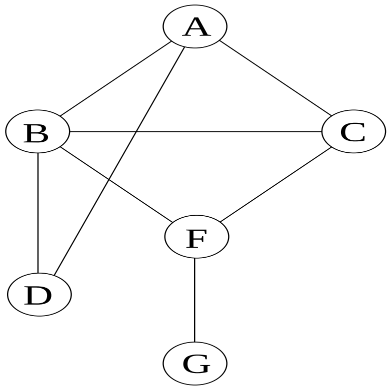

Figure 9.12: A constraint network example

of x in G_d. If B_y is the parent of B_x in the bucket-tree, then the separator of x and y is the set of variables appearing in $B_x \cap B_y$.

Example 9.3.2 Consider a constraint network defined over the graph in Figure 9.12. Figure 9.14(a) shows the initial buckets along the ordering $d = (A, B, C, D, F, G)$, and the ρ constraints that will be created and passed by adaptive-consistency from top to bottom. On its right, the figure displays the same computation as a message-passing along its bucket-tree. □

Theorem 9.3.3 *A bucket tree of a constraint network $\mathcal{R}$ is a tree-decomposition of $\mathcal{R}$.*

Proof. We need to provide mappings χ and ψ and show that tree-decomposition properties hold for a bucket tree:

1. $\chi(B_x)$ contains x and its earlier neighbors in the induced graph (G^*, d) along ordering d.

2. $\psi(B_x)$ contains all constraints whose highest-ordered argument is x.

By construction, the first tree-decomposition property holds. In order to prove connectedness, let's assume to the contrary that there are two buckets B_x and B_y, both containing variable z, but that on the path between B_x and B_y, there is a B_u that does not contain z. Let B_i (B_j) be the first bucket on the path from B_x (B_y) to B_u containing z, but its parent does not contain z. Because B_u is on the path between B_i and B_j, it must be that $i \neq j$. Since the parents of B_i and B_j do not contain z, it must be that variable

Algorithm Adaptive-Tree Consistency (ATC)
Input: A problem $\mathcal{R} = (X, D, C)$, ordering d.
Output: Augmented buckets containing the original constraints and all the ρ constraints received from neighbors in the bucket-tree.
A minimal subproblem for each bucket.
0. Pre-processing:
Place each constraint in the latest bucket, along d, that mentions a variable in its scope. Connect bucket B_x to B_y, $y < x$, if variable y is the latest earlier neighbor of x in the induced graph G_d.
1. Top-down phase: (AC)
For $i = n$ to 1, process bucket B_{x_i}:
Let $\rho_1, ..., \rho_j$ be all the constraints in B_{x_i} at the time B_{x_i} is processed, including original constraints of $\mathcal{R}$. The constraint $\rho_{x_i}^y$ sent from x_i to its parent y, is computed by

$$\rho_{x_i}^y(sep(x_i, y)) = \pi_{sep(x_i,y)} \bowtie_{i=1}^{j} \rho_i$$

2. Bottom-up phase:
For $i = 1$ to n, process bucket B_{x_i}:
Let $\rho_1, ..., \rho_j$ be all the constraints in B_{x_i} at the time B_{x_i} is processed, including the original constraints of $\mathcal{R}$. The constraints $\rho_{x_i}^{z_j}$ for each child bucket z_j is computed by

$$\rho_{x_i}^{z_j}(sep(x_i, z_j)) = \pi_{sep(x_i,z_j)}(\bowtie_i \rho_i)$$

Figure 9.13: Algorithm Adaptive-Tree Consistency (ATC)

z was eliminated at nodes B_i and B_j during the top-down phase of bucket-tree elimination. However, this is impossible, because during the top-down phase, each variable is eliminated exactly once. Therefore, B_u cannot exist. $\square$

Since the bucket-tree is a tree-decomposition, it can be processed by CTE. adds a bottom-up message passing and is equivalent to CTE. Algorithm *Adaptive Tree Consistency* (ATC) is given in Figure 9.13. In the top-down phase, each bucket receives constraint messages ρ from its children and sends a ρ constraint messages to its parent. This portion is equivalent to AC. In the bottom-up phase, each bucket receives a ρ constraint from its parent and sends ρ constraints to each child.

Example 9.3.4 Figure 9.15 shows a complete execution of ATC along the linear order of buckets and along the bucket-tree. The ρ constraints are viewed as messages placed on the outgoing arcs. $\square$

Since a bucket-tree is a tree-decomposition, and since it can be shown that CTE

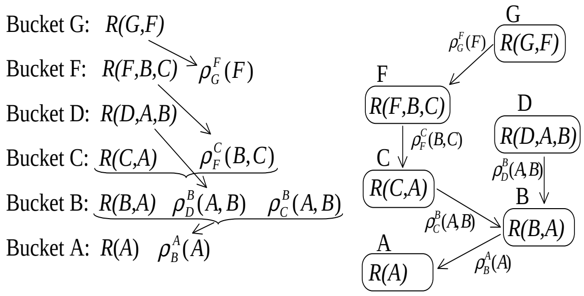

Figure 9.14: Execution of AC along the bucket-tree

applied to a bucket-tree is equivalent to ATC, then ATC both generates a backtrack-free representation along certain orderings and augments this with the generation of minimal subproblems. Furthermore,

Theorem 9.3.5 (Complexity of ATC) *Let w^* be the induced width of G along ordering d. The time complexity of ATC is $O(r \cdot deg \cdot exp(w^*))$, where deg is the maximum degree in the bucket-tree. The space complexity of ATC is $O(n \cdot exp(w^*))$.*

Proof: Since the number of buckets is n, from the analysis of CTE we can derive that the time complexity of ATC is $O((r+n) \cdot deg \cdot exp(w^*))$. Assuming that $r > n$, we get the desired bound for time complexity. Since the size of each message is $exp(sep)$, and since for bucket-trees the separator size is just w^*, we get space complexity of $O(n \cdot exp(w^*))$ as well. It is possible to modify ATC so that its time complexity will be bounded by $O(r \cdot exp(w^* + 1))$ only, without increasing space complexity. $\Box$.

9.4 Summary

The chapter present the class of acyclic constraint networks that provides a generalization of tree problems to non-binary constraints and show that such networks can be solved by a form of relational arc-consistency. Subsequently, it presents a tranformation, called Join-tree clustering (JTC) of any constraint netwwork into an acyclic one. The tansformation involves a graph-based step that embeds the constraint hypergraph in a hyper-tree also called a join-tree, which induces a collection of subproblems. The second step solves the subproblem concluding the creation of an acyclic problem.

Subsequently we present a unifying representation of tree-clustering that can accomodates many tree-based algorithms appearing in many areas of computation (probabilistic reasoning, combinatorial optimization) for defining a tree-decomposition and an associated cluster tree-elimination algorithm. The relationship with bucket-elimination

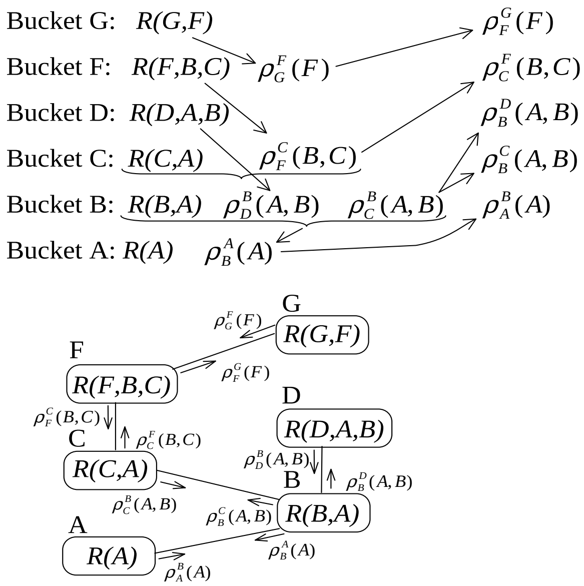

Figure 9.15: Propagation of ρ messages along the bucket-tree

algorithms is explicated. It is shown that a variable elimination algorithms can be viewed as message passing algorithms along specific tree-decompositions.

The time and space complexity of all algorithms is shown to be time exponential in the tree-width and hyper-width and space exponential in the separator of the tree-decomposition.

9.5 Chapter notes

Join-tree clustering was introduced in constraint processing by Dechter and Pearl [46] and in probabilistic networks by Spigelhalter et. al [108]. Both methods are based on the characterization by Relational-database researchers that acyclic-databases have an underlying tree-structure, called join-trees that allow polynomial query processing using join operations and easy identification procedures [12, 116, 162]. In both constraint networks and belief networks, it was observed that the complexity of compiling any knowledge-base into an acyclic one is exponential in the cluster size which is characterized by the induced-width or tree-width. At the same time, variable-elimination algorithms developed by [13], [182] and [45] (e.g., adaptive-consistency and bucket-elimination) were also observed to be governed by the same complexity graph-parameter and the similarity between the two approaches was analyzed [46]. Independently of this investigation, a parameter tree-width was under an intensive investigation in the theoretic graph-community. It characterizes the best embedding of a graph (or a hypergraph) in a hypertree. A variety of connections between hypertrees, chordal graphs, k-trees, were made by Arnborg and his colleagues [6, 145]. They showed that finding the smallest tree-width of a graph is NP-complete, but deciding if the graph has a tree-width below a certain constant k is polynomial in k.

In recent years, research has focused on a variety of greedy and other approximation algorithms for tree-width and induced-width [11, 156]. More recently a new parameter called hyper-width was introduced [70] and shown to provide a wider characterization of tractability also characterizing the embedding a hypergraph in a hyper-tree. This extension is valuable primarily for constraint networks. The tree-decomposition formulation is based on [70].

9.6 Exercises

1. Evaluate the complexity of algorithm PRIMAL-ACYCLICITY.
2. Consider the crossword puzzle described in Chapter 2.
   (a) Use join-tree clustering to transform the problem into an acyclic constraint network. What is the complexity of this transformation? Present explicitly the constraint problem generated.
   (b) Apply CTE (show the messages schematically).
   (c) Compare the above two approaches (tree-clustering, CTE) for solving the problem in terms of time and space complexity.
3. Consider the graph problem in Figure 9.7a and assume that all the constraints are binary "not-equal" constraints and all the domains of the variables have the values $\{1, 2\}$ except the domain of B is $\{1, 2, 3\}$. a. Generate a tree-decomposition. b. Apply CTE to this problem fully (show the messages exactly).
4. Consider a grid problem with 9 variables and binary constraints. Describe a join-tree decomposition created by JTC.
   (a) What is the tree-width, hyper-width and seperators of your decomposition.
   (b) What is the time and space complexity of CTE on the decomposition.
   (c) Show schematically how adaptive-tree-consistency will work on this problem. What would be the time and space complexity?
5. Show that algorithm ATC can be modified so that its time complexity is bounded by $O(r \cdot exp(w^* + 1))$ when r is number of constraints and w^* is the induced-width of the ordering ([155]).

Chapter 10

Hybrids of Search and Inference: Time-Space tradeoffs

Throughout this book two primary constraint processing schemes emerge - those based on *conditioning* or search, and those based on *inference* or *derivation*. Search in constraint satisfaction takes the form of depth-first backtracking, while inference is performed by variable-elimination and tree-clustering algorithms, or by bounded local consistency enforcing. In analogy to human problem solving, conditioning corresponds to guessing (a value of a variable), or reasoning by assumption. The problem is divided into subproblems, conditioned on the instantiation of a subset of variables, each of which should be solved. On the other hand, inference correspond to thinking and understating the problem better. Inference-based algorithms derive and record new information, generating equivalent problem representation that facilitates easier solution.

Both search and inference algorithms have their advantages and disadvantages. Inference-based algorithms are efficient when the problem is sparse; when the induced-width of the problem is low, but otherwise require substantial time and memory. Search algorithms require exponential time in the worst case, but they need very little memory. Moreover, search often exhibit a much better average performance. Due to their complemtary properties, the key to designing efficient algorithms is to combine inference-based and conditioning-based algorithms, and try to tailor the balance of inference and search to the problem instance. This may better utilize the benefit of each scheme and allow better performance guarantees, better space complexity and better overall average performance.

Indeed, due to its better average performance and minimal space requirements, backtracking is more popular than variable elimination for finding a single solution to constraint problems. Still, on problems with low width the latter techniques exploit the problems' structure, and they are can be more efficient than backtracking, as demonstrated empirically in Figure 10.1. The figure summarizes experiments over randomly