Tractable Structures for Constraint satisfaction problems

Rina Dechter
Bren School of Information and Computer Science
University of California, Irvine, CA 92697-3425
{*dechter*}@*ics.uci.edu*

September 8, 2005

1 Introduction

1.1 Inference, Search and Hybrids

Throughout the past few decades 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. Compared to human problem solving techniques, conditioning is analogous to guessing (a value of a variable), or reasoning by assumption. The problem is then divided into subproblems, conditioned on the instantiation of a subset of variables, each of which should be solved. On the other hand, inference corresponds to reinterpreting or making deduction from the problem at hand. Inference-based algorithms derive and record new information, generating equivalent problem representations that facilitate an easier solution.

Search and inference algorithms have their relative advantages and disadvantages. Inference-based algorithms are better at exploiting the independencies captured by the underlying constraint graph. They therefore provide a superior worst-case time-guarantee as a function of graph-based parameters. Unfortunately, any method that is time-exponential in the tree-width is also *space*-exponential in the tree-width and, therefore, not practical for dense problems.

Brute-force Search algorithms are structure-blind. They traverse the network's search space where each path represents a partial or a full solution. The linear structure of these search spaces hide the structural independencies displayed in the constraint graph and therefore they may not be as effective. In particular they lack useful performance guarantees. On the other hand search algorithms are flexible in their memory needs and can even operate with linear memory. Also search often exhibits a much better average performance than their worst-case bounds, when augmented with various heuristics and especially when looking for a single solution. Given their complementary properties, combining inference-based and conditioning-based algorithms may better utilize the benefit of each scheme and allow improved performance guarantees, reduced space complexity and improve average performance.

This chapter focuses on structure-driven constraint processing algorithms. We will start with inference algorithms and show that their performance is controlled by graph parameters such as tree-width, induced-width and hyper tree-width. We then show that hybrids of search and inference can be controlled by graph-based parameters such as cycle-cutset and w-cutset. Finally, we present the notion of AND/OR search spaces for exploiting independencies displayed in the constraint graph, leading to graph-based performance bounds for search using parameters such as the depth of the pseudo-tree, path-width and tree-width.

1.2 Constraint Networks and Tasks

A constraint problem is defined in terms of a set of variables taking values on finite domains and a set of functions defined over these variables. We denote variables or subsets of variables by uppercase letters (**e.g.**, X, Y, Z, S, $R\ldots$) and values of variables by lower case letters (**e.g.**, x, y, z, s). An assignment ($X_1 = x_1, \ldots, X_n = x_n$) can be abbreviated as $x = (\langle X_1, x_1\rangle, \ldots, \langle X_n, x_n\rangle)$ or $x = (x_1, \ldots, x_n)$. For a subset of variables S, D_S denotes the Cartesian product of the domains of variables in S. x_S and $x[S]$ are both used as the projection of $x = (x_1, \ldots, x_n)$ over a subset S. We denote functions by letters f, g, h etc., and the scope (set of arguments) of the function f by $scope(f)$.

A *constraint network* $\mathcal{R}$ consists of a finite set of *variables* $X = \{X_1, \ldots, X_n\}$, each associated with a *domain* of discrete values, $D_1, \ldots, D_n$ and a set of *constraints*, $\{C_1, \ldots, C_t\}$. Each of the constraints is expressed as a relation, defined on some subset of variables, whose tuples are all the simultaneous value assignments to the members of this variable subset that, as far as this constraint alone is concerned, are legal.[1] Formally, a constraint C_i has two parts: (1) the subset of variables $S_i = \{X_{i_1}, \ldots, X_{i_{j(i)}}\}$, on which it is defined, called a *constraint-scope*, and (2) a *relation*, R_i defined over S_i : $R_i \subseteq D_{i_1} \times \cdots \times D_{i_{j(i)}}$. The relation denotes all compatible tuples of D_{S_i} allowed by the constraint. Thus a constraint network $\mathcal{R}$ can be viewed as the triplet $\mathcal{R} = (X, D, C)$. The *scheme* of a constraint network is defined as its set of scopes, namely, $scheme(\mathcal{R}) = \{S_1, S_2, \ldots, S_t\}, S_i \subseteq X$.

DEFINITION **1 ((operations on constraints))** *Let R be a relation on a set S of variables, let $Y \subseteq S$ be a subset of the variables, and let Y_I be an instantiation of the variables in Y. We denote by $\pi_Y(R)$ the projection of the relation R on the subset Y; that is, a tuple over Y appears in $\pi_Y(R)$ if and only if it can be extended to a full tuple in R. Let R_{S_1} be a relation on a set S_1 of variables and let R_{S_2} be a relation on a set S_2 of variables. We denote by $R_{S_1} \bowtie R_{S_2}$ the natural join of the two relations. The join of R_{S_1} and R_{S_2} is a relation defined over $S_1 \cup S_2$ containing all the tuples t, satisfying $t[S_1] \in R_{S_1}$ and $t[S_2] \in R_{S_2}$.*

An assignment of a unique domain value to each member of some subset of variables is called an *instantiation*. An instantiation is said to satisfy a given constraint C_i if the partial assignment specified by the instantiation does not violate C_i. An instantiation is said to be *legal* or *locally consistent* if it satisfies *all* the (relevant) constraints of the network. A consistent instantiation of *all* the variables of a constraint network is called a *solution* of the network, and the set of all

[1] This does not mean that the actual representation of any constraint is necessarily in the form of its defining relation, but that the relation can, in principle, be generated using the constraint's specification without the need to consult other constraints in the network.

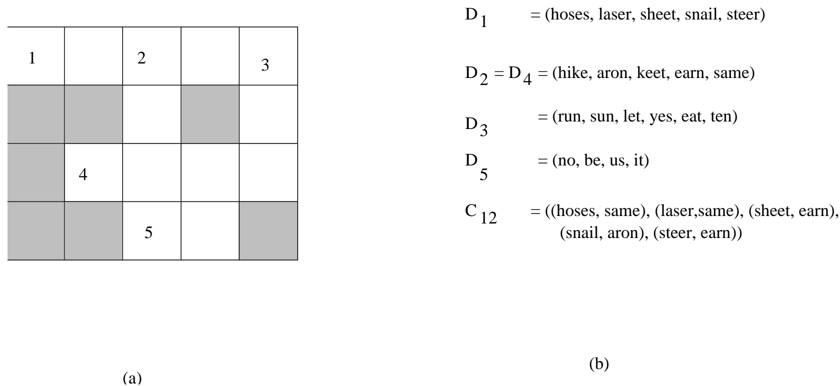


Figure 1: A crossword puzzle and its CN representation.

solutions is a relation, ρ, defined on the set of all variables. This relation is said to be *represented* by the constraint network. Formally,

$$\rho \;=\; \{x = (X_1 \;=\; x_1, \ldots, X_n \;=\; x_n) \mid \forall\, S_i \;\in\; scheme,\;\; \pi_{S_i} x \in R_i\}$$

. It can also be expressed as the join over all relations as $\rho = \Join_{R_i \in C} R_i$.

Example 1: Figure 1a presents a simplified version of a crossword puzzle (see *constraint satisfaction*). The variables are X_1 (1, horizontal), X_2 (2, vertical), X_3 (3, vertical), X_4 (4, horizontal), and X_5 (5, horizontal). The scheme of this problem is $\{X_1X_2, X_1X_3, X_4X_2, X_4X_3, X_5X_2\}$. The domains and some constraints are specified in Figure 1b. A tuple in the relation associated with this puzzle is the solution: $(X_1 = sheet, X_2 = earn, X_3 = ten, X_4 = aron, X_5 = no)$.

Typical tasks defined in connection with constraint networks are to determine whether a solution exists, to find one or all of the solutions, to count solutions or, when the problem is inconsistent, to find a solution that satisfies the maximum number of constraints (Max-CSP). Sometime, given a set of preferences over solutions defined via a cost function, the task is to find a consistent solution having maximum cost.

1.3 Graphical representations

Graphical properties of constraint networks were initially investigated through the class of *binary constraint networks* [15]. A *binary constraint network* is one in which every *constraint scope* involves at most two variables. In this case the network can be associated with a constraint graph, where each node represents a variable, and the arcs connect nodes whose variables are explicitly constrained. Figure 2 shows the constraint graph associated with the crossword puzzle in Figure 1.

A graphical representation of higher order networks can be provided by *hypergraphs*, where again, nodes represent the variables, and *hyperarcs* or *hyperedges* (drawn as regions) group variables that belong to the same scope. Two variations of this representation that can be used to facilitate structure-driven algorithms are *primal-constraint graph* and *dual-constraint graph*. A *Primal-constraint graph* (a generalization of the binary constraint graph) represents variables by nodes and associates an arc with any two nodes residing in the same constraint. A *dual-constraint-graph* represents each scope by a node (also called a *c-variable*) and associates a labeled arc with any two nodes whose scopes share variables. The arcs are labeled by the shared variables.

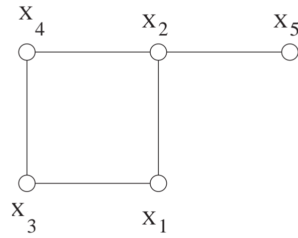


Figure 2: A constraint graph of the crossword puzzle.

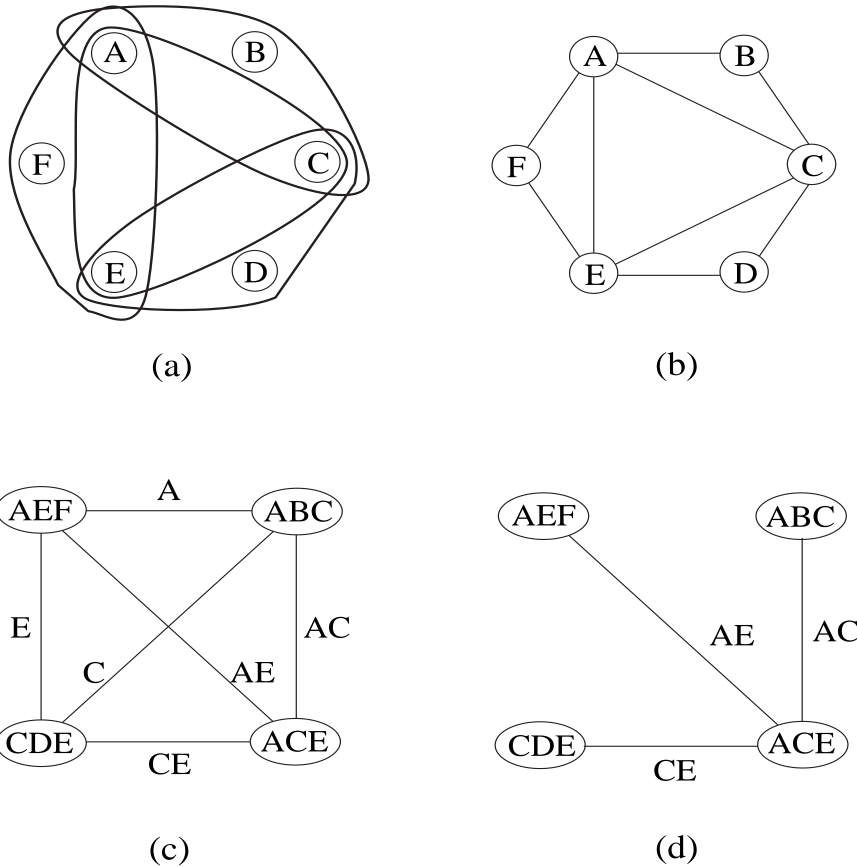


Figure 3: (a)Hyper, (b)Primal, (c)Dual and (d)Join-tree constraint graphs of a CSP.

For example, Figure 3 depicts the *hypergraph* (a), *primal* (b), and the *dual* (c) representations of a network with variables $A,\ B,\ C,\ D,\ E,\ F$ and constraints on the subsets (ABC),(AEF), (CDE) and (ACE). The constraints themselves are symbolically given by the inequalities: $A + B \leq C$, $A + E \leq F$, $C + D \leq E$, $A + C \leq E$, where the domains of each variable are the integers $[2, 3, 4, 5, 6]$.

The *dual* constraint graph can be viewed as a transformation of a nonbinary network into a special type of *binary* network: the domain of the c-variables ranges over all possible value combinations permitted by the corresponding constraints, and any two adjacent c-variables must obey the

restriction that their shared variables should have the same values (i.e., the c-variables are bounded by equality constraints). For instance, the domain of the the c-variable ABC is {224, 225, 226, 235, 236, 325, 326, 246, 426, 336} and the binary constraint between ABC and CDE is given by the relation: $R_{ABC,CDE} = \{(224, 415), (224, 426)\}$. Viewed in this way, any network can be solved by binary networks' techniques. Next we summarize graph concepts that will be used throughout the chapter.

DEFINITION 2 **(graph,hyper-graph)** *A* graph *is a pair $G = \{V, E\}$, where $V = \{X_1, \ldots, X_n\}$ is a set of vertices, and $E = \{(X_i, X_j)|X_i, X_j \in V\}$ is the set of edges (arcs). The degree of a variable is the number of arcs incident to it. A* **hyper-graph** *is a pair $H = (V, S)$ where $S = \{S_1, ..., S_t\}$ is a set of subsets of V, called hyper-edges.*

DEFINITION 3 **(primal graph, dual graph)** *The* **primal graph** *of a hyper-graph $H = (V, S)$ is an undirected graph $G = (V, E)$ such that there is an edge $(u, v) \in E$ for any two vertices $u, v \in V$ that appear in the same hyper-edge (namely, there exists S_i, s.t., $u, v \in S_i$). The* **dual graph** *of a hyper-graph $H = (V, S)$ is an undirected graph $G = (S, E)$ that has a vertex for each hyper-edge, and there is an edge $(S_i, S_j) \in E$ when the corresponding hyper-edges share a vertex ($S_i \cap S_j \neq \emptyset$).*

DEFINITION 4 **(hyper-tree)** *A hyper-graph is a* **hyper-tree***, also called* **acyclic hypergraph** *if and only if its dual graph has an edge subgraph (one that has the same set of vertices as the dual graph, but a subset of the edges) that is a tree and that satisfies the connectedness property, namely all the nodes in the dual graph that contain a common variable, form a connected subgraph.*

DEFINITION 5 **(induced-width)** *An* **ordered graph** *is a pair (G, d) denoted G_d where G is an undirected graph, and $d = (X_1, ..., X_n)$ is an ordering of the vertices. The* width of a vertex *in an ordered graph is the number of its earlier neighbors. The* width of an ordered graph, *$w(G_d)$, is the maximum width of all its vertices. The* **induced width of an ordered graph***, $w^*(G_d)$, is the width of the induced ordered graph, denoted G^*_d, obtained by processing the vertices recursively, from last to first; when vertex X is processed, all its earlier neighbors are connected. The* induced width of a graph*, $w^*(G)$, is the minimal induced width over all its orderings [11] It is well known that the induced width of a graph is identical to its* tree-width *[12].*

For various relationships between these and other graph parameters see [1, 18].

2 Structure-based tractability in Inference

Almost all the known structure-based techniques rely on the observation that *binary* constraint networks whose constraint graph is a *tree* can be solved in linear time [15, 25, 11]. The solution of tree-structured networks are discussed next, and later it is shown how they can be used to facilitate the solution of a general constraint network.

Tree-solving
Input: A tree network $T = (X, D, C)$.
Output: A backtrack-free network along an ordering d.
1. generate a width-1 ordering, $d = X_1, \ldots, X_n$.
2. **let** $X_{p(i)}$ denote the parent of X_i in the rooted ordered tree.
3. **for** $i = n$ to 1 **do**
4. *Revise* $((X_{p(i)}), X_i)$;
5. **if** the domain of $X_{p(i)}$ is empty, exit (no solution exists).
6. **endfor**

Figure 4: Tree-solving algorithm

2.1 Solving Tree-Networks

Given a tree-network over n variables (Fig. 5), the first step of the *tree-algorithm* is to generate a *rooted-directed* tree. Each node in this tree (excluding the root) has one *parent node* directed toward it and may have several *child* nodes, directed away from it. Nodes with no *children* are called *leaves*. An ordering, $d = X_1, X_2, \ldots, X_n$, is then enforced such that a parent always precedes its children. In the second step, the algorithm processes each arc and its associated constraint from leaves to root, in an orderly layered fashion. For each directed arc from X_i to X_j it removes a value from the domain of X_i if it has *no consistent match* in the domain of X_j. Finally, after the root is processed, a backtracking algorithm is used to find a solution along the ordering d.

It can be shown that the algorithm is linear in the number of variables. In particular, backtracking search, which in general is an exponential procedure, is guaranteed to find a solution without facing any dead-ends.

The tree algorithm is sketched in Figure 4. The *revise* procedure revise(X_j, X_i) remove any value from the domain of X_j that has no match in the domain of X_i. The complexity of the *tree-solving* algorithm is bounded by $O(nk^2)$ steps where k bounds the domain size, because the *revise* procedure, which is bounded by k^2 steps, is executed at most n times.

THEOREM **1** *[26] A binary tree constraint problem can be solve in $O(nk^2)$ when n is the number of variables and k bounds the domain size.*

2.2 Acyclic Networks

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 creation of a class of *acyclic constraint networks*.

As noted, any constraint network $\mathcal{R} = (X, D, C),\ C = \{R_{S_1}, ..., R_{S_t}\}$ 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 $H = \{S_1, ..., S_l\}$. The dual graph of a constraint hypergraph

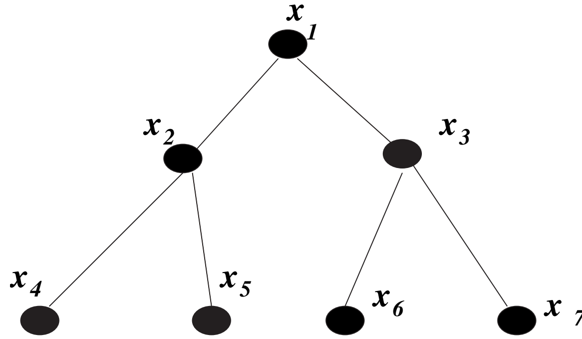


Figure 5: A tree network

associates a node with each constraint scope (or a hyperedge) and has an arc for each two nodes sharing variables. As noted before, this association facilitates the transformation of a non-binary constraint problem into a binary one, called the *dual problem*. Therefore, if a problem's dual graph happens to be a tree, it means that the dual constraint problem, can be efficiently solved 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, if 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. It is not normally easy to recognize redundant constraints. In the dual representation, however, 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) already enforces that equality. Removing such constraints does not alter the problem.

Example 1 Looking again at Figure 3, we see that the arc between (AEF) and (ABC) in Figure 3(c) is redundant because variable A also appears along the alternative path $(ABC) - AC - (ACE) - AE - (AEF)$. A consistent assignment to A is thereby 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 3(d).

We call the property that ensures such legitimate arc removal the *running intersection property* or *connectedness* property. The running intersection property can be defined over hypergraphs or over their dual graphs, and is used to characterize equivalent concepts such as *join-trees* (defined over dual graphs) or *hypertrees* (defined over hypergraphs). An *arc subgraph* of a graph contains the same set of nodes as the graph, and a subset of its arcs.

DEFINITION **6 (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 2 Considering again the graphs in Figure 3, we can see that the join-tree in Figure 3(d) satisfies the connectedness property. The hypergraph in Figure 3(a) has a join-tree and is therefore a hypertree.

An acyclic constraint network can be solved efficiently. Because the constraint problem has a join-tree, its dual problem is a tree of binary constraints and can therefore be solved by the tree-solving algorithm. Note that the domains of variables in the dual problem are bounded by the number of tuples in the input constraints. In Figure 6, 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 3 Consider the tree dual problem in Figure 3(d) and 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 $d = (R_{ACE}, R_{CDE}, R_{AEF}, R_{ABC})$. When processing R_{ABC}, 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: $R_{ACE} = \pi_{ACE}(R_{ACE} \bowtie R_{CDE}) = \{(0,1,1)\}$. A solution can then 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 then similarly extending the assignment to $F = 0$ and $B = 0$, to satisfy R_{AEF} and R_{ABC}.

Since the complexity of a tree-solving algorithm is $O(nk^2)$, where n is the number of variables and k bounds the domain size, the implied complexity of acyclic-solving 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, like so: 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 **2 (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, where r is the number of constraints and l bounds the number of tuples in each constraint relation.* □

Several efficient procedures for identifying acyclic networks and for finding a representative join-tree were developed in the area of relational databases [27]. One scheme that proved particularly useful is based on the observation that a network is acyclic if, and only if, its primal graph is both *chordal* and *conformal* [4]. A graph is *chordal* if every cycle of a length of at least four has a chord, i.e., an edge joining two nonconsecutive vertices along the cycle. A graph is *conformal* if each of its maximal *cliques* (i.e. subsets of nodes that are completely connected) corresponds to a

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 updated relations and 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 6: Acyclic-solving algorithm

constraint scope in the original constraint networks. The *chordality* of a graph can be identified via an ordering of the graph called the *maximal cardinality ordering*, (*m-ordering*); it always assigns the next number to the node having the largest set of already numbered neighbors (breaking ties arbitrarily).

It can be shown [33] that in an m-ordered chordal graph, the parents of each node must be completely connected. If, in addition, the maximal cliques coincide with the scopes of the original $\mathcal{R}$, both conditions for acyclicity would be satisfied. Because for chordal graphs each node and its parent set constitutes a clique, the maximal cliques can be identified in linear time, and then a *join tree* can be constructed by connecting each maximal clique to an ancestor clique with which it shares the largest set of variables [12].

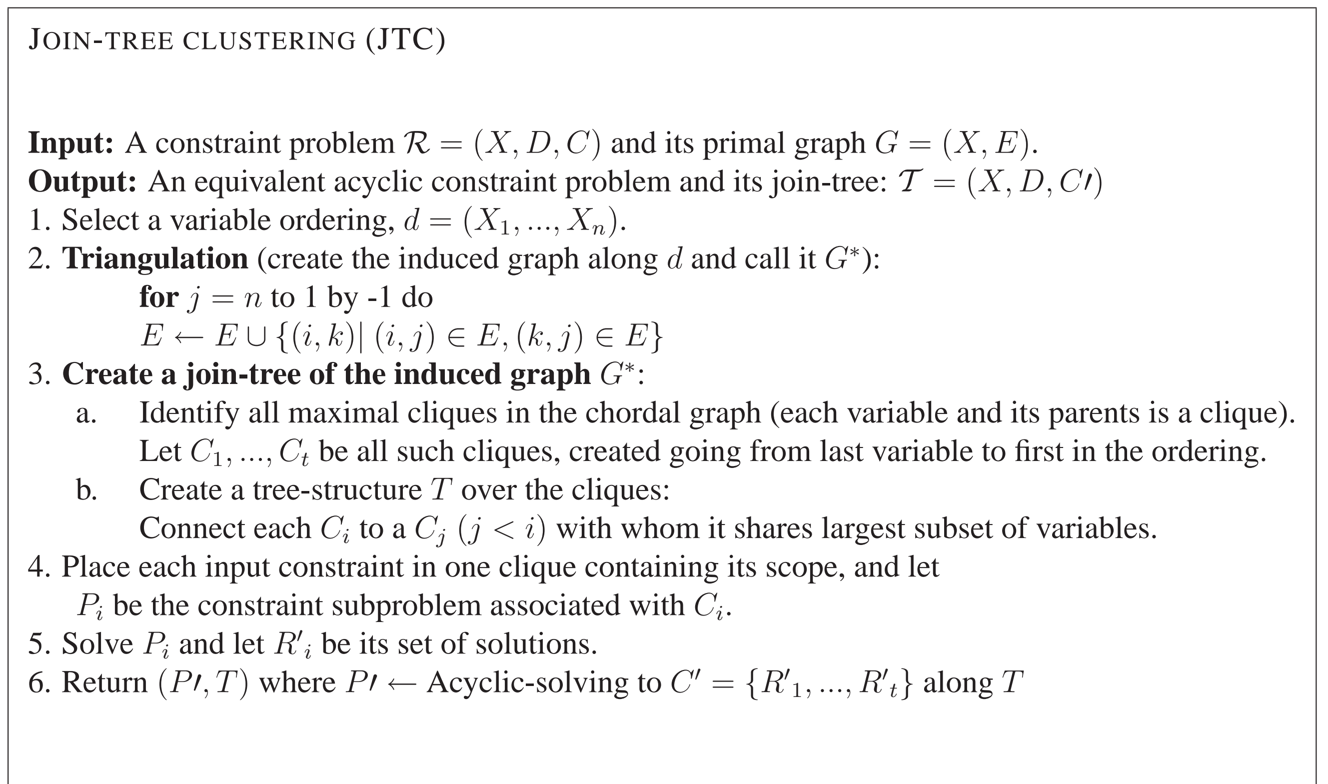
2.3 Tree-decompositions and tree-width

Since acyclic constraint networks can be solved efficiently, we naturally 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 transformation process is tractable the resulting algorithm is polynomial. This compilation process is called *join-tree clustering*.

The graphical input to the above scheme is the constraint hypergraph $\mathcal{H} = (X, H)$, where H is the set of scopes of the constraint network. Its output is a hypertree $\mathcal{S} = (X, S)$ and a partition of the original hyperedges into the new tree hyperedges defining the subproblems. Each subproblem is then solved, and its set of solutions is a new constraint whose scope is the hyperedge. Therefore, the result is a network having one constraint per hyperedge of the tree S, and, by construction, is acyclic.

2.3.1 Join-tree clustering and processing

There are various specific methods that decompose a hypergraph into a hypertree. The aim is to generate hypertrees having small-sized hyperedges because this implies small constraint subproblems. The most popular approach manipulates the constraint's primal graph and it emerges from the primal recognition process of acyclic networks described earlier. Since acyclic problems have primal graph that is chordal, the idea is to make the primal graph of a given network, which is not acyclic, chordal and then associates the maximal cliques of the resulting chordal graph with hyper-edges. Those hyperedges will be the new scope in the targeted acyclic problem. Given an ordered graph, chordality can be enforced by recursively connecting all parents of every node starting from the last node to the first. The procedure that generates the hypertree partitioning using the chordality algorithm and that then associates each cluster of constraints with its full set of solutions is called join-tree clustering (JTC) described in Figure 7.



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: $\mathcal{T} = (X, D, C\prime)$
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 its parents is a clique). Let $C_1, ..., C_t$ be all such cliques, created going from last variable to first in the ordering.
   b. Create a tree-structure T over the cliques:
   Connect each C_i to a C_j $(j < i)$ with whom it shares largest subset of variables.
4. Place each input constraint in one clique containing its scope, and let
   P_i be the constraint subproblem associated with C_i.
5. Solve P_i and let R'_i be its set of solutions.
6. Return $(P\prime, T)$ where $P\prime \leftarrow$ Acyclic-solving to $C' = \{R'_1, ..., R'_t\}$ along T

Figure 7: Join-tree clustering

The first three steps of algorithm JTC manipulate 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 edges). Step 5 solves each subproblem defined by a cluster, and thus creates one new constraint for each subproblem (clique). Step 6 applies the acyclic-solving algorithms. We will designate steps 5 and 6 as a separate procedure called *Join-Tree Processing (JTP)* which transforms a join-tree of constraint subproblems into an acyclic problem and then process it by acyclic-solving.

JOIN-TREE-PROCESSING (JTP)

Input: A collection of subproblems $\mathcal{P} = \{P_1, ..., P_n\}$ each $\mathcal{P}_i = \{R_{i_1}, ...R_{i_j}\}$ and a tree structure $T = (P, E)$.
Output: An equivalent pair-wise consistent acyclic constraint problem and its join-tree: $\mathcal{T} = (X, D, C\prime)$
1. Solve P_i and let R'_i be its set of solutions.
2. Return $(P\prime, T)$ where $\mathcal{P}\prime \leftarrow$ Acyclic-Solving to $C' = \{R'_1, ..., R'_t\}$, along T.

Figure 8: Join-tree processing

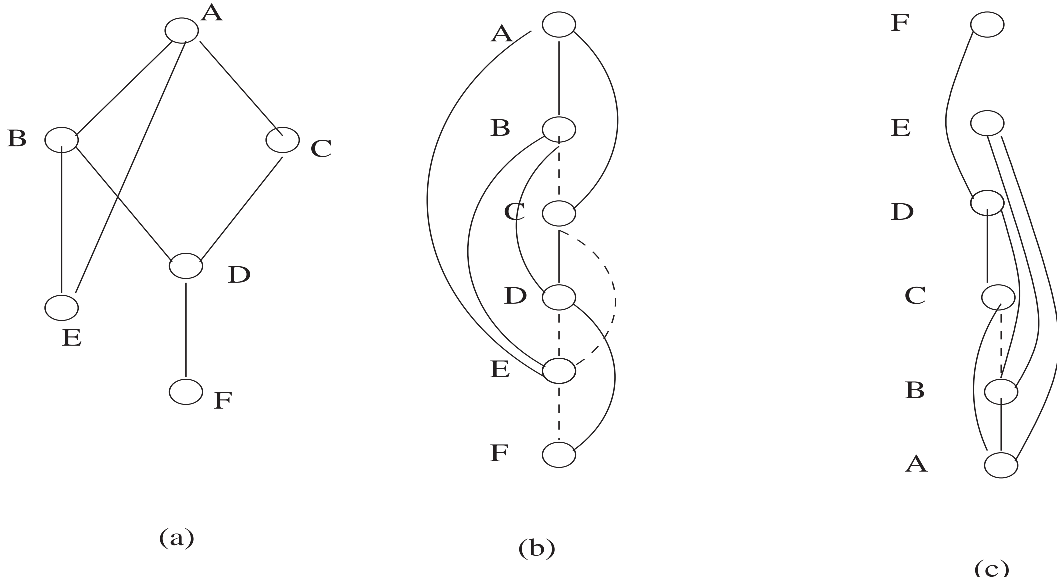


Figure 9: A graph (a) and two of its induced graphs (b) and (c). All arcs included.

Example 4 Consider the graph in Figure 9(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(b). Performing join- 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(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(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 10 (broken arcs are not part of the join-trees). Next, focusing on the join-tree in Figure 10b, JTC 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). Finally, algorithm JTP solves the subproblems P_1, P_2, P_3, P_4, and replace each with R_1, R_2, R_3, R_4, where R_i is the solution relation of P_i, yielding a desired acyclic network and its consistent solution.

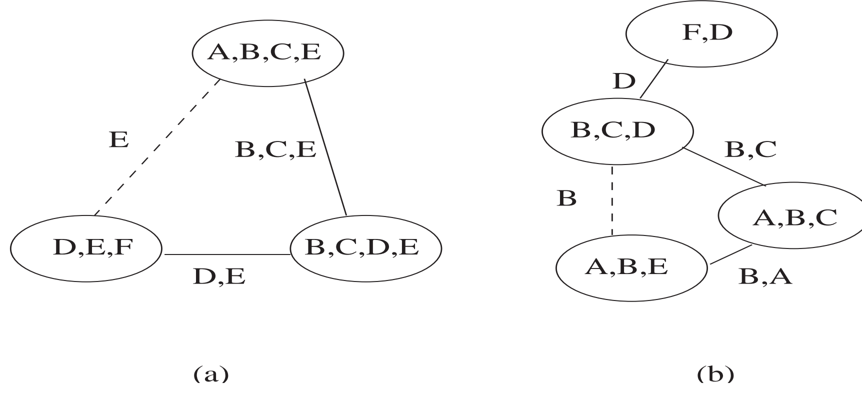


Figure 10: Join-graphs of the induced graphs from (a)Figure 9(b) and Figure 9(c). (All arcs included.) The corresponding join-trees are the same figures with the broken arcs removed.

THEOREM **3 (complexity of JTC)** *Given a constraint network having n variables and r constraints, the time and space complexity of join-tree clustering is $O(r \cdot w^*(d) \cdot logk \cdot k^{w^*(d)+1})$, where k is the maximum domain size and $w^*(d)$ is the induced width of the ordered graph.*

Proof: Finding a tree-decomposition of a hypergraph (Step 1 of JTC) is performed over the constraint primal graph requiring $O(n^2)$ steps. The most expensive step is Step 5, which computes 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 having at most $w^*(d)+1$ variables and r_i constraints costs $O(r_i \cdot exp(w^*(d)+1))$. Summing over all subproblems $\sum_i r_i exp(w^*(d)+1)$, yields the desired bound. Step 6 of acyclic-solving can be bounded by $O(n \cdot w^*(d) \cdot logk \cdot k^{w^*(d)+1})$, which is just applying acyclic-solving when $l = O(k^{w^*(d)+1})$. Summing the two components we get total bound of $O(r \cdot k^{w^*(d)+1} + r \cdot w^*(d) \cdot logk \cdot k^{w^*(d)+1}) = O((r \cdot w^*(d) \cdot logk) k^{w^*(d)+1})$ $\square$

2.3.2 General Tree-Decomposition Schemes

Algorithm Join-tree-clustering lumps together the structuring process of generating a tree of constraints with their processing. It also commits to a specific structuring algorithm that is based on chordal graphs.

In this section we reformalize the notion of a tree-decomposition and provide an alternative, time-space sensitive way, for its processing. This exposition unifiwes several related schemes such as variable elimination, join-tree clustering and hyper-tree decomposition.

DEFINITION **7 (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 $scope(R_i) \subseteq \chi(v)$.*

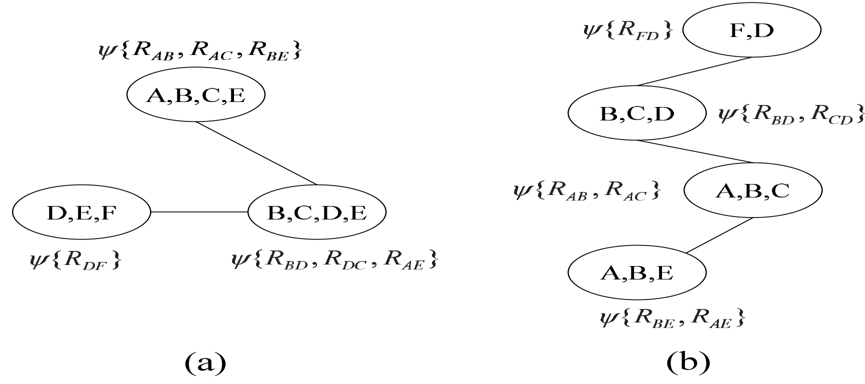


Figure 11: Two tree-decompositions

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 **8 (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, the separator of u and v is defined as $sep(u, v) = \chi(u) \cap \chi(v)$.*

Example 5 Consider the binary constraint problem whose primal graph appears in Figure 9(a). The join-trees in Figure 10(a) and (b) were obtained via triangulation in ordering 9b and 9c and can be redescribed in Figure 11, using the two labeling functions described above.

Graphically, a tree-decomposition defines a hypertree embedding of a hypergraph. The smallest tree-width among all such embeddings is the *tree-width* of the constraint hypergraph. Once a tree-decomposition is available, algorithm *Cluster-Tree Elimination (CTE)* in Figure 12, can processes the decomposition.

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, and generate their join projected on the separator. The resulting constraint is then sent to v (remember that v is adjacent to u in the tree). It is important to note that the particular implementation of equation 1 in CTE can vary. One option is to generate the combined relation ($\bowtie_{R_i \in cluster(u)} R_i$) before sending messages to each neighbor. This will yield a processing algorithm similar to JTP. The other option, which we assume here, is that the message sent to each neighbor is created without recording the relation ($\bowtie_{R_i \in cluster(u)} R_i$). Rather, each tuple is projected on the separator immediately after being created. This will yield a better memory utilization as we will show shortly.

The output of CTE 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 is a *minimal subproblem* relative to the input constraint problem $\mathcal{R}$. Intuitively, a subproblem of a constraint network is minimal if you can correctly answer any query on it without having to refer back to information in the whole network. More precisely, a subproblem over a

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

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

$$m_{(u,v)} = \pi_{sep(u,v)}(\bowtie_{R_i \in cluster(u)} R_i) \quad (1)$$

endfor
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 12: Algorithm cluster-tree elimination (CTE)

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 (decomposable 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 $\mathcal{R}$.*

Convergence is guaranteed, but it may take as long as the diameter of the tree in the worst case when messages are not ordered. 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.

Example 6 *Figure 13 shows the messages propagated for the join-tree in Figure 11b. Since cluster 1 contains only one relation, the message from cluster 1 to 2 is the projection of R_{FD} over the separator between cluster 1 and 2, which is variable D. The message $m_{(2,3)}$ from cluster 2 to cluster 3 joins the relations in cluster 2 with the message $m_{(1,2)}$, and projects over the separator between cluster 2 and 3, which is $\{B, C\}$, and so on.*

Since CTE can be shown to be equivalent to generating and solving an acyclic constraint problem by a tree-solving algorithm it is clearly sound.

2.3.3 Complexity of CTE

It is well known that given an induced graph having an induced-width w, it implies a tree-decomposition having tree-width w and vice versa. Thus, from now on we will use w for both

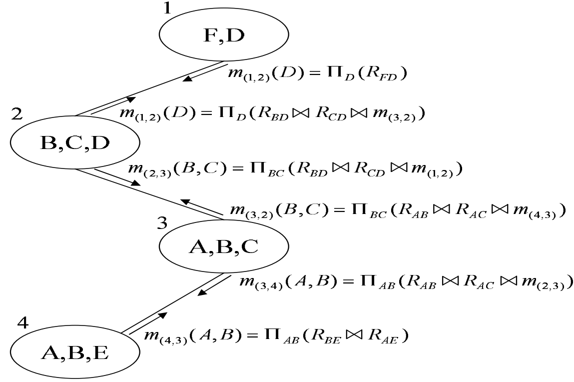


Figure 13: Example of messages sent by CTE

.

induced-width and tree-width of a given tree decomposition, while w^* for the minimal tree-width/induced-width of a graph.

Computing the messages. Algorithm CTE can be subtly varied to influence its time and space complexities. If we first record the joined relation in Equation 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 thereby make the space complexity identical to the size of the sent constraint message. The message can be computed 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 **4 (Time-space complexity)** *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 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 $O(N \cdot k^{sep})$.*

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 algorithm 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 recorded constraint 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 [31, 21]. □

Join-tree clustering as tree-decomposition. Algorithm JTC is a specific algorithm for creating the tree-decomposition. Because it generates the full set of solutions for each node, its space complexity is exponential in the tree-width w^*, unlike CTE whose 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 $O(r \cdot exp(w^*))$. Clearly, this dis-

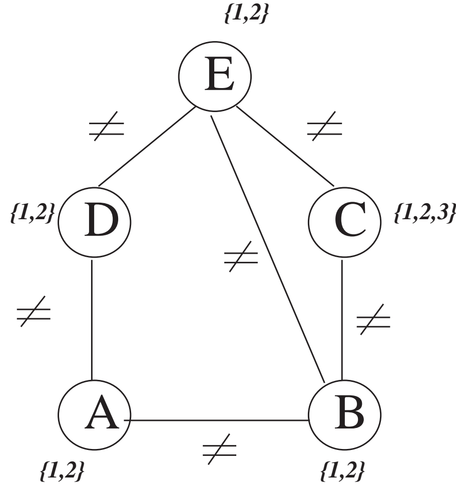


Figure 14: A graph coloring example

tinction matters only if there is a substantial difference between the tree-width and the maximum separator size of a given tree-decomposition. See [19] for more details.

2.4 Variable-Elimination schemes

We next show that variable-elimination algorithms such as Adaptive-consistency [11] can be viewed as message passing in a CTE type algorithm. Adaptive consistency, described in Figure 16, works by eliminating variables one by one, while deducing the effect of the eliminated variable on the rest of the problem. Adaptive-consistency can be described using the bucket data-structure. Given a variable ordering $d = A, B, D, C, E$ in a graph coloring example depicted in Figure 14 we process the variables from last to first, namely, from E to A. Step one is to partition the constraints into *ordered buckets*. All the constraints mentioning the last variable E are put in a bucket designated as $bucket_E$. Subsequently, all the remaining constraints mentioning D are placed in $bucket_D$, and so on. The initial partitioning of the constraints is depicted in Figure 15a. In general, each constraint is placed in the bucket of its latest variable.

After this initialization step, the buckets are processed from last to first. Processing bucket E produces the constraint $D = C$, which is placed in bucket C. By processing bucket C, the constraint $D \neq B$ is generated and placed in bucket D. While processing bucket D, we generate the constraint $A = B$ and put it in bucket B. When processing bucket B inconsistency is discovered. The buckets' final contents are shown in Figure 15b. The new inferred constraints are displayed to the right of the bar in each bucket.

At each step the algorithm generates a reduced but equivalent problem with one less variable expressed by the union of unprocessed buckets. Once the reduced problem is solved its solution is guaranteed to be extendible to a full solution since it accounted for the deduced constraints generated by the rest of the problem. Therefore, once all the buckets are processed, and if there are no inconsistencies, a solution can be generated in a backtrack-free manner. Namely, a solution is assembled progressively assigning values to variables from the first variable to the last. A value of the first variable is selected satisfying all the current constraints in its bucket. A value for the second variable is then selected which satisfies all the constraints in the second bucket, and so on. Process-

$Bucket(E)$: $E \neq D, E \neq C$
$Bucket(C)$: $C \neq B$
$Bucket(D)$: $D \neq A,$
$Bucket(B)$: $B \neq A,$
$Bucket(A)$:

(a)

$Bucket(E)$: $E \neq D, E \neq C$
$Bucket(C)$: $C \neq B \ || \ D = C$
$Bucket(D)$: $D \neq A, \ ||, D \neq B$
$Bucket(B)$: $B \neq A, \ || \ B = A$
$Bucket(A)$: $||$

(b)

Figure 15: A schematic execution of adaptive-consistency

Algorithm Adaptive consistency (AC)
1. **Input:** A constraint problem $R_1, ...R_t$, ordering $d = X_1, ..., X_n$.
2. **Output:** An equivalent backtrack-free set of constraints and a solution.
3. **Initialize:** Partition constraints into $bucket_1, ...bucket_n$. $bucket_i$ contains all relations whose scope include X_i but no higher indexed variable.
4. **For** $p = n \ downto \ 1$, process $bucket_p$ as follows

for all relations $R_1, ...R_m$ defined over $S_1, ...S_m \in bucket_p$ do
(Find solutions to $bucket_p$ and project out X_p:)
$A \leftarrow \bigcup_{j=1}^{m} S_j - \{X_i\}$

$R_A \leftarrow R_A \cap \ \Pi_A(\bowtie_{j=1}^{m} R_j)$

5. If R_A is not empty, add it to the bucket of its latest variable.
Else, the problem is inconsistent.

6. Return $\cup_j bucket_j$ and generate a solution: for $p = 1$ to n do
assign a value to X_p that is consistent with previous assignments and satisfies all the constraints in $bucket_p$.

Figure 16: Algorithm Adaptive consistency

ing a bucket amounts to solving a subproblem defined by the constraints appearing in the bucket, and then restricting the solutions to all but the current bucket's variable. Adaptive-consistency is an instance of a general class of variable elimination algorithms called bucket-elimination that are applicable across many tasks [8].

The complexity of adaptive-consistency is linear in the number of buckets and in the time to process each bucket. Since processing a bucket amounts to solving a constraint subproblem (the computation in a bucket can be described in terms of the relational operators of *join* followed by

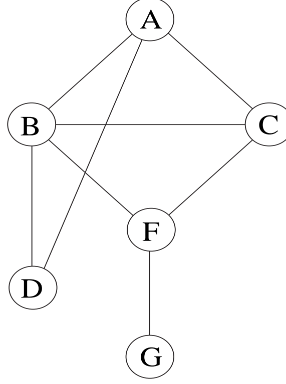


Figure 17: A constraint network example

projection) its complexity is exponential in the number of variables mentioned in a bucket which is bounded by the *induced-width* of the constraint graph along that ordering [11].

THEOREM **5 (Complexity of AC)** *Let w^* be the induced width of G along ordering d. The time and space complexity of AC is $O(r \cdot exp(w^* + 1))$.*

2.5 Adaptive-consistency as tree-decomposition

We now show that adaptive-consistency can be viewed as a message-passing algorithm along a bucket-tree, which is a special case of tree-decomposition. 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 the set of buckets, each contains those constraints in C whose latest variable in d is X_i. A *bucket-tree* of $\mathcal{R}$ in an ordering d, has buckets as its nodes, and bucket B_X is connected to bucket B_Y if the constraint generated by adaptive-consistency in bucket B_X 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 $B_{Z_1}, ...B_{Z_t}$.

It is easy to see that a bucket tree of $\mathcal{R}$ is a tree-decomposition of $\mathcal{R}$ where for bucket B_X, $\chi(B_X)$ contains X and its earlier neighbors in the induced graph along ordering d, while $\psi(B_X)$ contains all constraints whose highest-ordered argument is X. Therefore,

THEOREM **6** *A bucket tree of a constraint network $\mathcal{R}$ is a tree-decomposition of $\mathcal{R}$.*

A bucket-tree can be processed by CTE or by JTP. Thus we can add a bottom-up message passing to adaptive-consistency yielding *Adaptive Tree Consistency* (ATC) given in Figure 18. In the top-down phase, each bucket receives constraint messages ρ from its children and sends ρ constraint messages to its parent. This portion is identical to AC. In the bottom-up phase, each bucket receives a ρ constraint from its parent and sends a ρ constraint to each child.

Example 7 Consider a constraint network defined over the graph in Figure 17. Figure 19 left 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. Figure 20 shows a complete execution of ATC along the linear order of buckets and along the bucket-tree. The ρ constraints are displayed as messages placed on the outgoing arcs.

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.
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

$$\rho_{X_i}^{Y}(sep(X_i, Y)) = \pi_{sep(X_i,Y)} \bowtie_{i=1}^{j} \rho_i \tag{2}$$

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

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

Figure 18: Algorithm Adaptive-Tree Consistency (ATC)

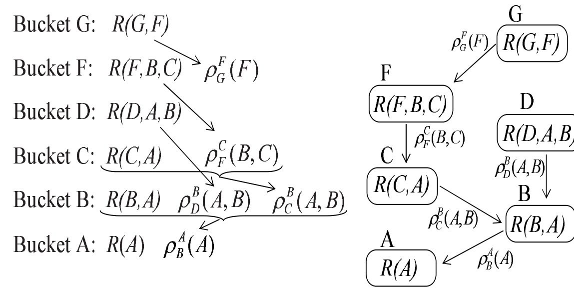


Figure 19: Execution of AC along the bucket-tree

THEOREM **7 (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 $O(n \cdot exp(w^*))$.*

Since the separators size is very close to the induced-width, there is no memory saving in ATC and it is better to employ JTP to process the bucket-tree or some other time-improving versions of CTE as discussed in [20]. The "deg" factor in the time complexity can therefore be removed yielding $O(r \cdot exp(w^*))$.

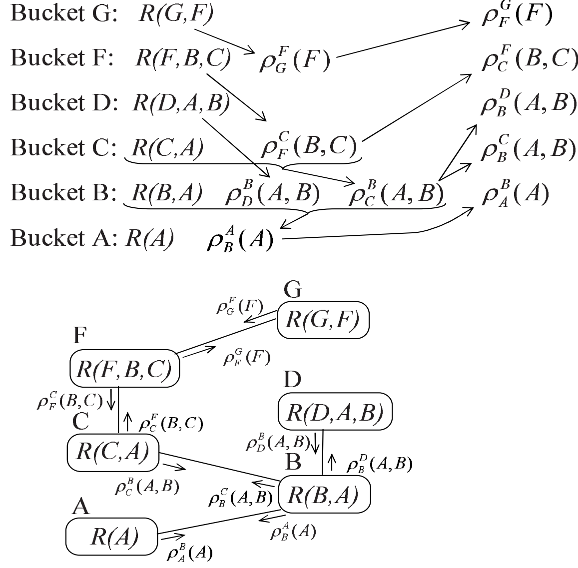


Figure 20: Propagation of ρ messages along the bucket-tree

2.6 Hyper-tree Decomposition

One problem with the tree-width in identifying tractability is that they are sensitive only to the primal constraint graph and not to its hypergraph structure. For example, an acyclic problem whose constraint's scope have high arity would have a high tree-width even though it can be processed in quadratic time in the input. A different graph parameter that is more sensitive to the hyper-graph structure is the hyperwidth [17]. It relies on a notion of hyper-tree decompositions for Constraint Satisfaction and it provides a stronger indicator of tractability than the tree-width.

DEFINITION **10 (hyper-tree decomposition)** *[17] A (complete) hyper-tree decomposition of a hyper-graph $HG = (X, S)$ is a triple $< T, \chi, \psi >$, where $T = (V, E)$ is a rooted tree, and χ and ψ are labelling functions which associate with each vertex $v \in V$ two sets $\chi(v) \subseteq X$ and $\psi(v) \subseteq S$, and which satisfies the following conditions:*

1. *For each edge $h \in S$, there exists $v \in V$ such that $h \in \psi(v)$ and $scope(h) \subseteq \chi(v)$ (we say that v strongly covers h).*
2. *For each variable $X_i \in X$, the set $\{v \in V | X_i \in \chi(v)\}$ induces a (connected) subtree of T.*
3. *For each $v \in V$, $\chi(v) \subseteq scope(\psi(v))$.*
4. *For each $v \in V$, $scope(\psi(v)) \cap \chi(T_v) \subseteq \chi(v)$, where $T_v = (V_v, E_v)$ is the subtree of T rooted at v and $\chi(T_v) = \cup_{u \in V_v} \chi(u)$.*

The hyper-width hw of a hyper-tree decomposition is $hw = max_v |\psi(v)|$.

A hyper-tree decomposition of a constraint network $\mathcal{R}$ is a hyper-tree-decomposition of its hyper-graph where the vertices are the variables of $\mathcal{R}$ and the scopes of constraints are the hyper-edges. The hyper-tree decomposition can be processed by JTP or by CTE and its complexity can be using the hyperwidth of the hypertree decomposition.

Processing hypertree decomposition by join-tree processing, JTP: Once a hyper-tree decomposition is available, 1. join all the relations in each cluster, yielding a single relation on each cluster. This step takes time and space $O((m+1) \cdot t^{hw})$ where t bounds the relation size and m is the number of hyperedges in the hyper-tree decomposition, and it creates an acyclic constraint satisfaction problem. 2. Process the acyclic problem by arc-consistency. This step can be accomplished in time $O(m \cdot hw \cdot t^{hw} \cdot logt)$ because there are m arc in the hyper-tree decomposition, each has at most $O(t^{hw})$ tuples so acyclic-solving is $O(mt^{hw}log(t^{hw})$ which yields the desired bound. We can summarize,

THEOREM **8** *[16] Let m be the number of hyper-edges in the hyper-tree decomposition of a constraint network $\mathcal{R}$, hw be its hyper-width and t be a bound on the relation size. A hyper-tree decomposition of a constraint problem can be processed by JTP in time $O(m \cdot hw \cdot logt \cdot t^{hw})$ and in space $O(t^{hw})$.*

Notice that there are tree-decompositions that are not hyper-tree decompositions as in 10, because hyper-tree decompositions require that variables labeling a vertex will be contained in the combined scope of its labeling functions (Condition 3 of Definition 10). This is not required by the tree-decomposition definition. For example, consider a single n-ary constraint R. It can be mapped into a bucket-tree with n vertices. Node i contains variables $\{1, 2, ...i\}$ but no constraints, except that node n contains also the input function. Both join-tree and hyper-tree decomposition will allow just one vertex that include the function and all its variables.

Processing hyper-tree decompositions by CTE: Recall that given a hyper-tree decomposition, each node u has to send a single message to each neighbor v. We can compute $m_{(u,v)}$ in the space saving mode as follows. 1., Combine all functions $\psi(u)$ in node u yielding function $h(u)$, namely, $h(u) = \bowtie_{R \in \psi(u)} R$. This step can be done in time and space $O(t^{|\psi(u)|})$. 2. For each neighbor c of u, $c \neq v$ iterate, $h(u) \leftarrow h(u) \bowtie m_{(c,u)}$ This step can be accomplished in $O(deg \cdot hw \cdot logt \cdot t^{hw})$ time and $O(t^{hw})$ space. 3. $m_{(u,v)} \leftarrow \pi_{\chi(u) \cap \chi(v)} h(u)$. We can summarize:

THEOREM **9** *A hyper-tree decomposition of a reasoning problem can be processed by CTE in time*

$$O(m \cdot deg \cdot hw \cdot logt \cdot t^{hw})$$

and space $O(t^{hw})$, where m is the number of edges in the hyper-tree decomposition, hw its hyper-width, and t is a bound on the size of the relational representation of each function in $\mathcal{R}$.

Theorem 9 does not apply for general tree-decompositions **??** because the complexity analysis assumed Condition 3 of Definition 10 (which means that every variable in a vertex of a tree must be covered by a function in that node). We can overcome this problem by thinking of all uncovered variables in a node as having a universal relation with the variables as its scope. In this case we can show

THEOREM **10** *A tree-decomposition of a constraint network $\mathcal{R}$ can be processed by CTE in time*

$$O(m \cdot deg \cdot hw^* \cdot logt \cdot t^{hw^*})$$

where t is a bound on the relation size, $hw^*(v) = (|\psi(v)| + |\{X_i | X_i \notin scope(\psi(v)\}|)$, and $hw^* = \max_{v \in V} hw^*(v)$.

Proof: Once we add the universal relation on uncovered variables we have a restricted hyper-tree decomposition to which we can apply the bound of Theorem 9 assuming the same implementation of CTE. The number of uncovered variables in a node v is $n(v) = |\{X_i | X_i \notin scope(\psi(v)\}|$. So the processing of a node takes time $O(t^{hw} \cdot k^{n(v)})$ where k bounds the domain size, yielding $O((max(t,k)^{hw^*})$. Assuming that $t > k$ we can use the time and space bound $O(t^{hw^*})$. Consequently, message passing between all nodes yields overall complexity as in Theorem 9 when hw is replaced by hw^*. $\Box$

2.7 Summary

This section discussed inference algorithms that transform a general constraint problem into a tree of constraints which can be solved efficiently. The complexity of the transformation process is exponentially bounded by the tree-width (or induced-width) of the constraint graph. It is also exponentially bounded by the hyperwidth of the hypertree-decomposition. Thus both the induced-width and tree-width and hyperwidth can be used to define structure-based tractable classes. Yet, the hyper-width defines a larger tractability class because every problem with a bounded tree-width has a bounded hyperwidth but not vice-versa.

3 Trading Time and Space by Hybrids of Search and Inference

As we noted at the introduction, search and inference have complementary properties. Inference exploit the graph structure and therefore allows structure-based time guarantees but require substantial memory. Search, does not posses good complexity time bounds yet it can operate in linear space. Therefore, using a hybrid of search and inference allows structure-driven tradoff of space and time. We next present two approaches for hybrids.

3.1 The cycle-cutset and w-cutset schemes

The algorithm s presented in this section exploit the fact that variable instantiation changes the effective connectivity of the constraint graph. Consider a constraint problem whose graph is given in Figure 21a. For this problem, instantiating x_2 to some value, say a, renders the choices of values to x_1 and x_5 independent, as if the pathway $x_1 - x_2 - x_5$ were blocked at x_2. Similarly, this instantiation blocks dependency in the pathway $x_1 - x_2 - x_4$, leaving only one path between any two variables. In other words, given that x_2 was assigned a specific value, the "effective" constraint graph for the rest of the variables is shown in Figure 21b. Here, the instantiated variable x_2 and its incident arcs are first deleted from the graph, and x_2 subsequently is duplicated for each of its neighbors. The constraint problem having the graph shown in Figure 21(a) when $x_2 = a$ is identical to the constraint problem having the graph in 21(b) with the same assignment $x_2 = a$.

In general, when the group of instantiated variables constitutes a cycle-cutset; a set of nodes that, once removed, would render the constraint graph cycle-free, the resulting conditional network is a tree (as shown in Figure 21b), and can be solved by the inference-based *tree-solving* algorithm. Note that in most practical cases it would take more than a single variable to cut all the cycles in the graph. Thus, a general way of solving a problem whose constraint graph contains cycles is

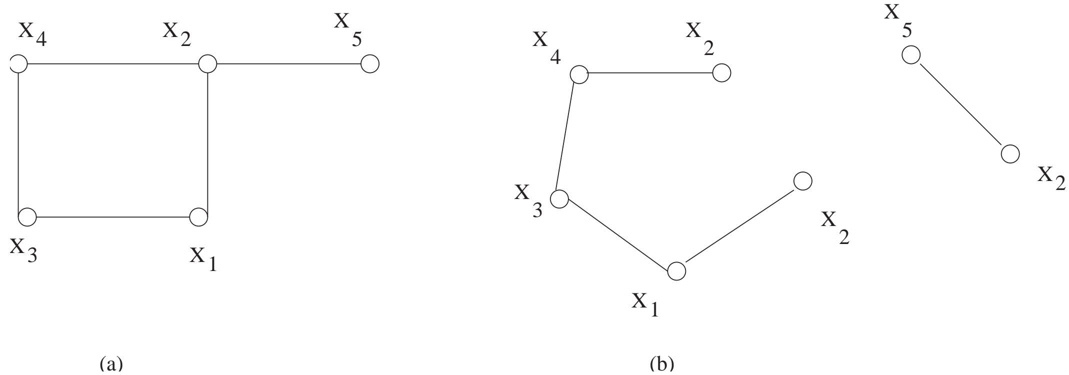


Figure 21: An instantiated variable cuts its own cycles.

to identify a subset of variables that cut all cycles in the graph, find a consistent instantiation of the variables in the cycle-cutset, and then solve the remaining problem by the *tree algorithm*. If a solution to this restricted problem (conditioned on the cycle-cutset values) is found, then a solution to the entire problem is at hand. If not, another instantiation of the cycle-cutset variables should be considered until a solution is found. If the task is to solve a constraint problem whose constraint graph is presented in Figure 21a, (assume x_2 has two values $\{a, b\}$ in its domain), first $x_2 = a$ must be assumed, and the remaining tree problem relative to this instantiation, is solved. If no solution is found, it is assumed that $x_2 = b$ and another attempt is made.

The number of times the tree-solving algorithm needs to be invoked is bounded by the number of partial solutions to the cycle-cutset variables. A small cycle-cutset is, therefore, desirable. However, since finding a minimal-size cycle-cutset is computationally hard, it will be more practical to settle for heuristic compromises. One approach is to incorporate this scheme within backtracking search. Because *backtracking* works by progressively instantiating sets of variables, we only need to keep track of the connectivity status of the constraint graph. As soon as the set of instantiated variables constitutes a cycle-cutset, the search algorithm is switched to the tree-solving algorithm on the restricted problem, i.e., either finding a consistent extension for the remaining variables (thus finding a solution to the entire problem) or concluding that no such extension exists (in which case backtracking takes place and another instantiation tried).

Example 8 Assume that backtracking instantiates the variables of the CSP represented in Figure 22a in the order C, B, A, E, D, F (Figure 22b). Backtracking will instantiate variables C, B and A, and then, realizing that these variables cut all cycles, will invoke a tree-solving routine on the rest of the problem: the tree-problem in Figure 22c with variables C, B and A assigned, should then be attempted. If no solution is found, control returns to backtracking which will go back to variable A.

The cycle-cutset scheme can be generalized. Rather than insisting on conditioning a subset (cutset) that cuts all cycles and yields width-1 subproblems, we can allow cutsets that create subproblems whose induced-width is higher than 1 but still bounded. This suggests a framework of hybrid algorithms parameterized by a bound w on the induced-width of subproblems solved by inference.

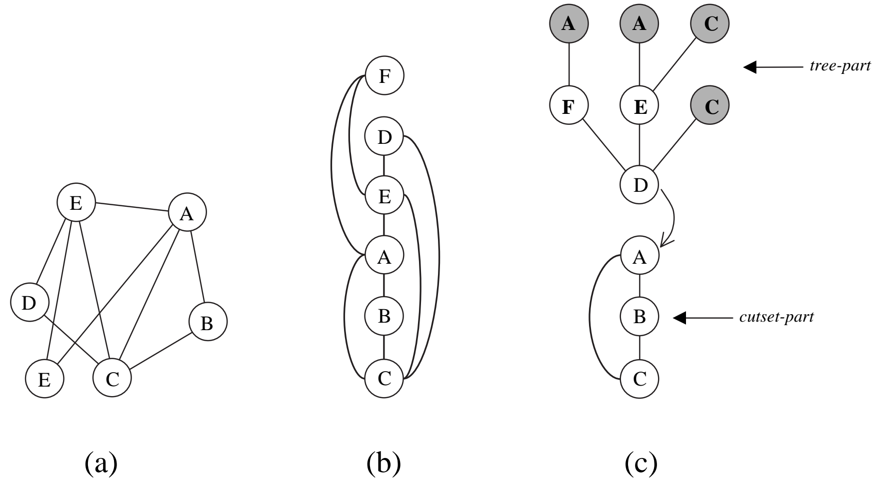


Figure 22: (a) a constraint graph (b) its ordered graph (c) The constraint graph of the cutset variable and the conditioned variable, where the assigned variables are darkened.

DEFINITION **11 (w-cutset)** *Given a graph G, a subset of nodes is called a w-cutset iff when the subset is removed the resulting graph has an induced-width less than or equal to w. A minimal w-cutset of a graph has a smallest size among all w-cutsets of the graph. A cycle-cutset is a 1-cutset of a graph.*

Finding a minimal w-cutset is a hard task, however, like in the special case of a cycle-cutset we can settle for a w-cutset relative to the given variable ordering. We can look for an initial set of the ordering that is a w-cutset. Then a backtracking algorithm can traverse the search space of the w-cutset and for each of its consistent assignment solve the rest of the problem by adpative-consistency.

Algorithm cutset-decomposition(w) (called elim-cond in [9]) is described in Figure 23. It runs backtracking search on the w-cutset and adaptive-consistency on the remaining variables. The constraint problem $\mathcal{R} = (X, D, C)$ conditioned on an assignment $Y = \bar{y}$ and denoted by $\mathcal{R}_{\bar{y}}$ is $\mathcal{R}$ augmented with the unary constraints dictated by the assignment $\bar{y}$. In the worst-case, all possible assignments to the w-cutset variables need to be tested. If c is the w-cutset size, k^c is the number of w-bounded subproblems needed to be solved, each requiring $O((n-c)k^{w+1})$ steps by inference, yielding,

THEOREM **11** *Algorithm cutset-decomposition(w) has time complexity of $O(n \cdot k^{c+w+1})$ where n is the number of variables, c is the w-cutset size and k is the domain size. The space complexity of the algorithm is $O(k^w)$.* $\square$

For the special case of cycle-cutset we get the cycle-cutset decomposition algorithm whose time complexity is $O((n-c)k^{c+2})$ and it operates in linear space.

Thus, the constant w controls the balance between search and variable-elimination, and thus affect the tradeoff between time and space.

Another approach for using the w-cutset principle is to alternate conditioning and the variable-elimination operation. Given a variable ordering for adaptive-consistency we can apply the variable elimination as long as the induced-width of the variables does not exceed w. If a variable

Algorithm cutset-decomposition(w)
Input: A constraint network $\mathcal{R} = (X, D, C)$, $Y \subseteq X$ which is a w-cutset. d is an ordering that starts with Y such that the adjusted induced-width, relative to Y along d, is bounded by w, $Z = X - Y$.
Output: A consistent assignment, if there is one.

1. **while** $\bar{y} \leftarrow$ next partial solution of Y found by backtracking, **do**
   (a) $\bar{z} \leftarrow adaptive - consistency(\mathcal{R}_{Y=\bar{y}})$.
   (b) **if** $\bar{z}$ is not *false*, return solution $(\bar{y}, \bar{z})$.
2. **endwhile**.
3. **return:** the problem has no solutions.

Figure 23: Algorithm *cutset-decomposition(w)*

has induced-width higher than w, it will be conditioned upon. The algorithm alternates between conditioning and elimination. This scheme was used both for solving SAT problems and for optimization tasks [28, 22]. Clearly, a cutset found using the alternating algorithm is a w-cutset and therefore can also be used within the cutset-decomposition scheme.

The cutset-decomposition scheme and the alternating cutset-elimination algorithm calls for a new optimization task on graphs:

DEFINITION **12 (finding a minimal w-cutset)** *Given a graph $G = (V, E)$ and a constant w, find a smallest subset of nodes U, such that when removed the resulting graph has induced-width less than or equal w.*

Finding a minimal w-cutset is hard but various greedy heuristic algorithms were investigated empirically. Several greedy and approximation algorithms for the special case of cycle-cutset can be found in the literature. The more general task of finding a minimal w-cutset was addressed in recent papers [14, 6] both for the cutset-decomposition version and for the alternating version. Note that for a given constant w, verifying that a given subset of nodes is a w-cutset can be accomplished in polynomial time (in the number of nodes), by deleting the candidate cutset from the graph and verifying that the remaining graph has an induced width bounded by w which is $O(exp(w))$.

In summary, the parameter w can be used within the cutset-decomposition scheme to control the trade-off between search and inference. If $w \geq w^*_d$, where d is the ordering used by cutset-decomposition(w), the algorithm coincides with adaptive-consistency. As w decreases, the algorithm requires less space and more time. It can be shown that the size of the smallest cycle-cutset (1-cutset), c_1 and the smallest induced width, w^*, obey the inequality $c_1 \geq w^* - 1$. Therefore, $1 + c_1 \geq w^*$, where the left side of this inequality is the exponent that determines the time complexity of cutset-decomposition(w=1), while w^* governs the complexity of bucket-elimination. In general,

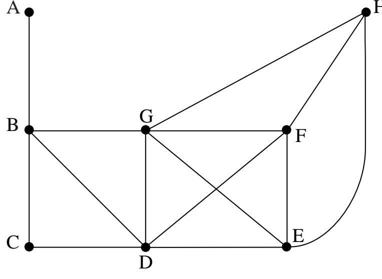


Figure 24: a primal constraint graph

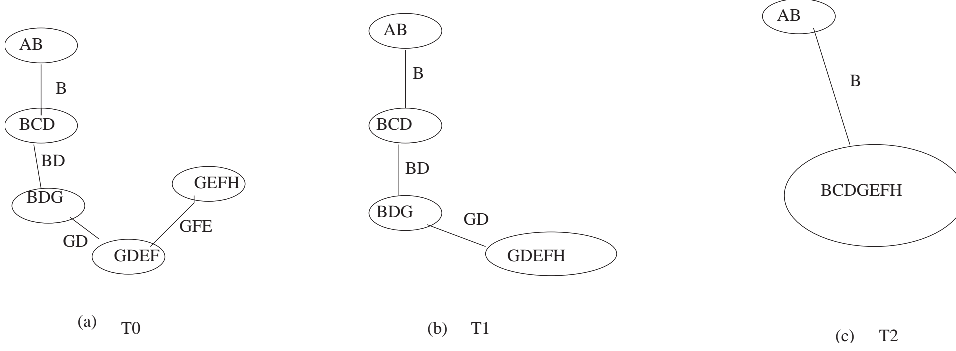


Figure 25: A tree-decomposition with separators equal to (a) 3, (b) 2, and (c) 1

$$1 + c_1 \geq 2 + c_2 \geq ...b + c_b, ... \geq w^* + c_{w^*} = w^*$$

But, since by definition $c_{w^*} = 0$, we get a hybrid scheme whose time complexity decreases as its space increases until it reaches the induced-width.

3.2 The super-bucket and super-cluster schemes; separator-width

We now present an orthogonal approach for combining search and inference. The inference algorithm CTE that process a tree-decomposition already contains a hidden combination of variable elimination and search. It computes functions on the separators using variable elimination and is space exponential in the separator's size. The clusters themselves can be processed by search in time exponential in the cluster size. Thus, one can trade even more space for time by allowing larger cliques but smaller separators.

Assume a problem whose tree-decomposition has tree-width r and maximum separator size s. Assume further that our space restrictions do not allow the necessary $O(exp(s))$ memory required when applyng CTE on such a tree. One way to overcome this problem is to combine the nodes in the tree that are connected by large separators into a singlecluster. The resulting tree-decomposition has larger subproblems but smaller separators. This idea suggests a sequence of tree-decompositions parameterized by the sizes of their separators as follows.

Let T be a tree-decomposition of hypergraph $\mathcal{H}$. Let $s_0, s_1, ..., s_n$ be the sizes of the separators in T, listed in strictly descending order. With each separator size s_i we associate a secondary

tree decomposition T_i, generated by combining adjacent nodes whose separator sizes are strictly greater than s_i. We denote by r_i the largest set of variables in any cluster of T_i, and by hw_i the largest number of constraints in T_i. Note that as s_i decreases, both r_i and hw_i increase. Clearly, from Theorem 4 it follows that,

THEOREM **12** *Given a tree-decomposition T over n variables, separator sizes $s_0, s_1, ..., s_t$ and secondary tree-decompositions having a corresponding maximal number of nodes in any cluster, $r_0, r_1, ..., r_t$. The complexity of CTE when applied to each secondary tree-decompositions T_i is $O(n \cdot exp(r_i))$ time, and $O(n \cdot exp(s_i))$ space (i ranges over all the secondary tree-decomposition).*

We will call the resulting algorithm SUPER-CLUSTER TREE ELIMINATION(s), or $SCTE(s)$. It takes a primary tree-decomposition and generates a tree-decomposition whose separator's size is bounded by s, which is subsequently processed by CTE. In the following example we assume that a naive-backtracking search processes each cluster.

Example 9 Consider the constraint problem having the constraint graph in Figure 24. The graph can be decomposed into the join-tree in Figure 25(a). If we allow only separators of size 2, we get the join tree T_1 in Figure 25(b). This structure suggests that applying CTE takes time exponential in the largest cluster, 5, while requiring space exponential in 2. If space considerations allow only singleton separators, we can use the secondary tree T_2 in Figure 25(c). We conclude that the problem can be solved either in $O(k^4)$ time (k being the maximum domain size) and $O(k^3)$ space using T_0, or in $O(k^5)$ time and $O(k^2)$ space using T_1, or in $O(k^7)$ time and $O(k)$ space using T_2.

Superbuckets. Since as we saw in Section 2.5, bucket-elimination algorithms can be extended to bucket-trees and since a bucket-tree is a tree-decomposition, by merging adjacent buckets we generate a *super-bucket-tree* (SBT) in a similar way to generating super clusters. This implies that in the top-down phase of bucket-elimination several variables are eliminated at once(see [9]).

Algorithm SCTE suggests new graph parameters.

DEFINITION **13** *Given a graph G and a constant s find a tree-decomposition of G having the smallest induced-width, w_s^*. Or, find a hyper-tree decomposition having the smallest hyperwidth denotes hw_s^**

Finding w_s^* is hard but it is easy for the special case of $s = 1$ as we show next.

3.2.1 Decomposition into non-separable Ccomponents

A special tree-decomposition occurs when all the separators are singleton variables. This type of tree-decomposition is attractive because it requires only linear space. While we generally cannot find the best tree-decompositions having a bounded separators' size in polynomial time, this is a feasible task when the separators are singletons. To this end, we use the graph notion of *non-separable components* [13].

DEFINITION **14 (non-separable components)** *A connected graph $G = (V, E)$ is said to have a separation node v if there exist nodes a and b such that all paths connecting a and b pass through v. A graph that has a separation node is called* separable*, and one that has none is called* non-separable. *A subgraph with no separation nodes is called a* non-separable component *or a* bi-connected component.

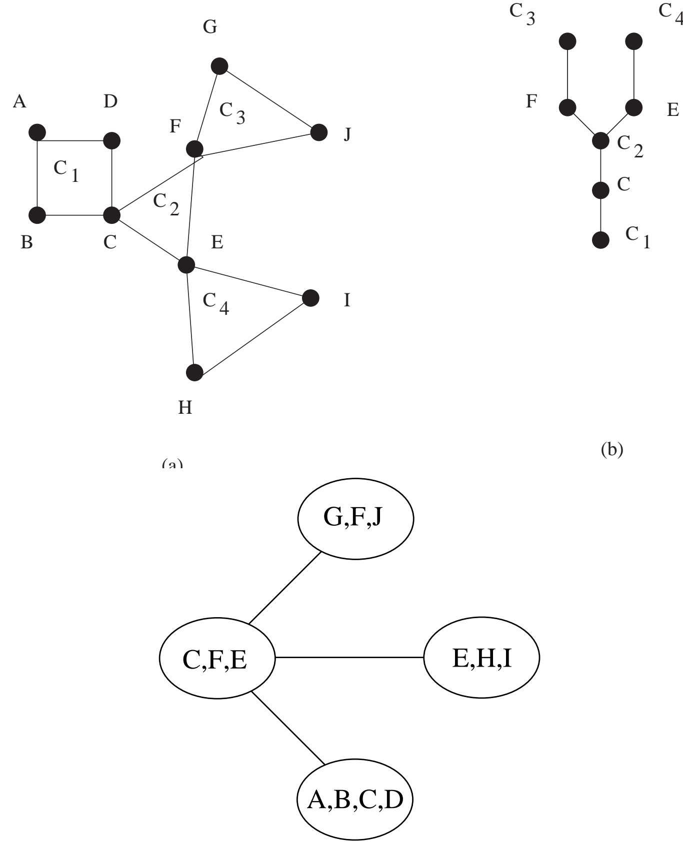


Figure 26: A graph and its decomposition into non-separable components.

An $O(\mid E \mid)$ algorithm exists for finding all the non-separable components and the separation nodes. It is based on a depth-first search traversal of the graph. An important property of non-separable components is that that they are interconnected in a tree-structured manner [13]. Namely, for every graph G there is a *tree* SG, whose nodes are the non-separable components $C_1, C_2, \ldots, C_r$ of G. The separating nodes of these trees are $V_1, V_2, \ldots, V_t$ and any two component nodes are connected through a separating node vertex in SG. Clearly the tree of non-separable components suggests a tree-decomposition where each node corresponds to a component, the variables of the nodes are those appearing in each component, and the constraints can be freely placed into a component that contains their scopes. Applying CTE to such a tree requires only linear space, but is time exponential in the components' sizes (see [9]).

Example 10 Assume that the graph in Figure 26(a) represents a constraint network having unary, binary and ternary constraints as follows:
$\mathcal{R} = \{R_{AD}, R_{AB}, R_{DC}, R_{BC}, R_{GF}, D_G, D_F, R_{EHI}, R_{CFE}\}$. The non-separable components and their tree-structure are given in Figure 26(b,c).

THEOREM **13 (non-separable components)** *If* $\mathcal{R} = (X, D, C)$ *is a constraint network whose constraint graph has non-separable components of at most size* r*, then the super-cluster-tree elimination algorithm, whose buckets are the non-separable components, is time exponential* $O(n \cdot exp(r))$ *and is linear in space.*

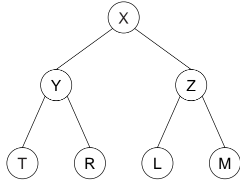


Figure 27: Tree T

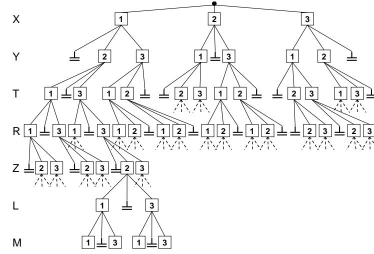


Figure 28: OR search tree

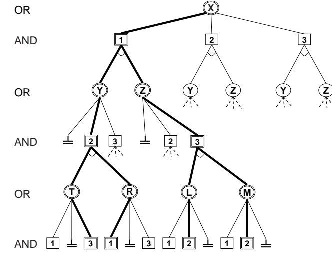


Figure 29: AND/OR search tree

3.2.2 Hinge decomposition

The non-separable component principle can be applied to the dual graph rather than to the primal constraint graph. Better yet, since the dual graph may contain redundant edges, we can first try to remove those edges to obtain a minimal dual graph (also called minimal join-graph) and then generate a tree of non-separable components. This idea is very related to another tree-decomposition principle proposed in the literature called hinge-decomposition [24]. Indeed a best hinge decomposition can be obtained in polynomial time, yielding smallest component in a bi-component tree decomposition of the dual graph whose some redundant arcs are removed.

4 Structure-based tractability in search

Search algorithms typically traverse the problem's space, where each path representing a partial or a full solution. Their main virtue is that they can operate using bounded memory. However, the main drawback is that the linear structure of the search space hides the independencies of the constraint network. Next we show that defining *AND/OR search spaces* can overcome this difficulty because the AND/OR principle displays the independencies in the constraint graph and can therefore become exponentially smaller than the traditional search space (called OR space). As a result, search algorithms can have graph-based performance guarantees like inference schemes.

4.1 AND/OR Search Trees

DEFINITION **15 (AND/OR search tree based on DFS tree)** *Given a constraint network $\mathcal{R}$ and a DFS spanning tree T of its primal graph, the AND/OR search tree of $\mathcal{R}$ based on T, denoted S_T, has alternating levels of OR nodes (labeled with variable names, e.g. X) and AND nodes (labeled with variable values, e.g. $\langle X, v\rangle$). The root of S_T is an OR node labeled with the root of T. The children of an OR node X are AND nodes, each labeled with a value of X, $\langle X, v\rangle$. The children of an AND node $\langle X, v\rangle$, are OR nodes, labeled with the variables that are children of X in T. A solution is a subtree containing the root node and for every OR node, it includes one of its child nodes and for every AND nodes it includes all its children.*

Consider the tree T in Fig. 27 describing a graph coloring problem over domains $\{1, 2, 3\}$. Its traditional OR search tree along the DFS ordering $d = (X, Y, T, R, Z, L, M)$ is given in Fig. 28, its AND/OR search tree based on the DFS tree T and a highlighted solution subtrees are given in Fig. 29.

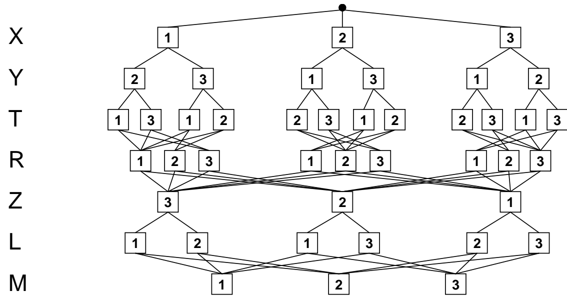


Figure 30: Minimal OR search graph of the tree problem in Fig. 27

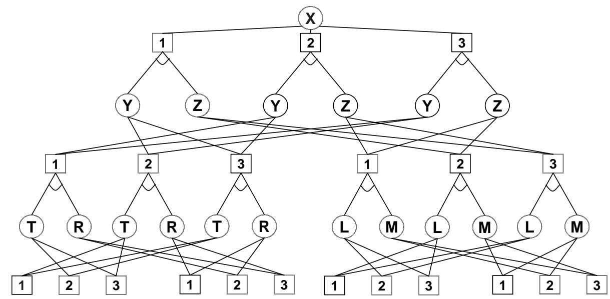


Figure 31: Minimal AND/OR search graph of the tree problem in Fig. 27

The construction of AND/OR search trees can be guided not just DFS spanning trees but also by *pseudo-trees* which include DFS trees [26, 2]. Pseudo-trees have the property that every arc of the constraint graph is a back-arc in the pseudo-tree (i.e. it doesn't connect across different branches). Clearly, any DFS tree and any chain are pseudo-trees. It is easy to see that searching an AND/OR tree guided by a pseudo-tree T is exponential in the depth m of T. Also, it is known that if A GRAPH HAS A TREE-WIDTH W^* it also has a pseudo-tree whose depth M satisfies $m \leq w^* logn$ [2]. It is easy to see that,

THEOREM **14 (complexity parameter)** *Given a constraint network $\mathcal{R}$ and a pseudo-tree T, its AND/OR search tree S_T is sound and complete (contains all and only solutions) and its size is $O(n \cdot \exp(m))$ where m is the depth of its backbone pseudo-tree. A constraint network that has a tree-width w^* has an AND/OR search tree whose size is $O(\exp(w^* \cdot \log n))$.*

Backjumping algorithms [9] are backtracking search applied to the regular OR space, which uses the problem structure to jump back from a dead-end as far back as possible. In *graph-based backjumping* (GBJ) each variable maintains a graph-based induced ancestor set which ensures that no solutions are missed by jumping back to its deepest variable. Graph-based backjumping extracts knowledge about dependencies from the constraint graph alone. Whenever a dead-end occurs at a particular variable X, the algorithm backs up to the most recent variable connected to X in the graph. It can be shown that backjumping searching for a single solution, in effect explores an AND/OR search space. Indeed, when *backjumping* is performed on a DFS ordering of the variables, its complexity can be bounded by $O(exp(m))$ steps, m being the depth of the DFS tree.

4.2 AND/OR Search Graphs

It is often the case that certain states in the search tree can be merged because the subtrees they root are identical. Any two such nodes are called *unifiable*, and when merged, transform the search tree into a search graph. For example, in Fig. 29, the search trees below the paths $\langle X, 2\rangle$, $\langle Y, 1\rangle$ and $\langle X, 3\rangle$, $\langle Y, 1\rangle$ are identical, so the corresponding nodes are unifiable.

In general, merging all the unifiable subtrees given an AND/OR search graph yields a unique graph, called the *minimal AND/OR search graph*. Merging is applicable to the traditional OR search space as well. However, in many cases it will not be able to reach the compression we see in the AND/OR search graph. Fig. 30 and Fig. 31 show a comparison between minimal OR and AND/OR search graphs for the problem in Fig. 27.

In some cases identifying unifiable nodes is easy. The idea is to extract from each path only the relevant *context* that completely determines the unexplored portion of the space. Subsequently, the subgraph is only solved once and the results are indexed by the context and cached. It can be shown that, (see [10] for details) that

THEOREM **15** *Given G, a pseudo-tree T and its induced width w the size of the minimal AND/OR search graph based on T is $O(n \cdot k^w)$, when k bounds the domain size.*

We can show that the minimal AND/OR search graph is bounded exponentially by the primal graph's tree-width while the OR minimal search graph is bounded exponentially by its path-width. It is well known [18] that for any graph $w^* \leq pw^* \leq w^* \cdot \log n$. It is also easy to place m^* (the minimal pseudo-tree depth) yielding $w^* \leq m^* \leq pw^* \leq w^* \cdot \log n$.

Searching the AND/OR graphs rather than the AND/OR tree can be shown to be related to recording no-goods during backtracking search.

5 Summary and Bibliographical Notes

Structure-based tractability. Throughout this chapter several techniques were presented that exploit the structure of the constraint network. Several graph parameters stood out in the analysis. The two main classes are *width*-based and *cutset*-based. Width-based parameters capture the size of clusters required to make the graph a tree of clusters. These include the *tree-width* also known as *induced-width* w^*, (appearing in *adaptive-consistency, tree-clustering* and in searching AND/OR graphs using caching of goods and no-goods, which is also known as *constraint recording* in dependency-directed backtracking). It also includes path-width (pw) which captures the cluster size required to embed the graph in a chain of clusters, and the hyperwidth hw appearing in the hypertree decomposition which captures the number of constraints in a tree of clusters. Cutset-based parameters include the *cycle-cutset size* c_1 and more generally the *i-cutset size* c_i (appearing in the cutset-decomposition method and which capture the number of variable that need to be removed from the constraint graph to make its tree-width bounded by i. This concept can be extended in an obvious way to cutset-hyper-decomposition to capture cutsets that make the graph have a bounded hyperwidth, rather than width. Another parameter that does not belong to the above two classes is the *depth of a DFS-tree and a pseudo-tree* m (appearing when searching AND/OR trees and in backjumping). The *size of largest non-separable component* r_1 (appearing in the decomposition to bi-connected components), the size of hinges (appearing in bi-connected decomposition of a minimal dual graph) and more generally the size or $sperator$-based tree-width r_s appearing in SCTE method were also discussed and belong width-based parameters that capture time-space tradeoffs.

It is well known [18, 2] that for any graph $w^* \leq m^* \leq pw^* \leq w^* \cdot \log n$. Relating width-based parameters to cutset parameters we have that $w^* \leq c_i + i$ holds. Also graphs having bounded tree-width have bounded hyperwidth but not vice-versa. Therefore the hyperwidth is the most informative parameter capturing tractability. However, verifying that a graph has a bounded tree-width below w can be done in polynomial time, while verifying bounded hyperwidth is not known to be polynomial. In addition, a given tree-decomposition may have smaller tree-width than hyperwidth. When memory is bounded we can use SCTE(i) or cutset-decomposition(i) for an appropriate i so that memory of $O(exp(i)$ is feasible.

Bibliographical notes. Join-tree clustering was introduced in constraint processing by Dechter and Pearl [12] and in probabilistic networks by Spigelhalter et. al [23]. Both methods are based on the characterization by relational-database researchers that acyclic-databases have an underlying tree-structure, called join-tree, that allows polynomial query processing using join-project operations and easy identification procedures [4, 27, 33]. 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 [5], [30] and [11] (e.g., adaptive-consistency and bucket-elimination) were also observed to be governed by the same complexity graph-parameter. The similarity between the two approaches from the constraint perspective was analyzed [12]. Independently of this investigation, the tree-width parameter was undergoing intensive investigation in the theoretic-graph-community. It characterizes the best embedding of a graph (or a hypergraph) in a hypertree. Various connections between hypertrees, chordal graphs and k-trees were made by Arnborg and his colleagues [1, 29]. 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. A more recent analysis is given by Bodlaender [7].

The decomposition into hinges was presented in [24]. aAs noted any hinge-decomposition is closely related to bi-component tree decomposition of the dual graph whose redundant arcs are removed. The parameter hyper-width was introduced by Gottlob, Leone and Scarcello [16] and shown to provide the most inclusive characterization of tractability. In recent years, research has focused on a variety of greedy and other approximation algorithms for tree-width and induced-width [3, 32].

References

[1] S. A. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25:2–23, 1985.

[2] R. Bayardo and D. Miranker. A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In *AAAI'96: Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, 1996.

[3] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *Uncertainty in AI (UAI'96)*, pages 81–89, 1996.

[4] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database ochemes. *Journal of the ACM*, 30(3):479–513, 1983.

[5] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[6] B. Bidyuk and R. Dechter. On finding w-cutset in bayesian networks. In *Uncertainty in AI (UAI04)*, 2004.

[7] H.L. Bodlaender. Treewidth: Algorithmic techniques and results. In *MFCS-97*, pages 19–36, 1997.

[8] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[9] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[10] R. Dechter and R. Mateescu. The impact of and/or search spaces on constraint satisfaction and counting. In *Proceeding of Constraint Programming (CP2004)*, pages 731–736, 2004.

[11] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.

[12] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, pages 353–366, 1989.

[13] S. Even. Graph algorithms. In *Computer Science Press*, 1979.

[14] Myan Fishelson and Dan Geiger. Optimizing exact genetic linkage computations. *RECOMB*, pages 114–121, 2003.

[15] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[16] Nicola Leone Georg Gottlob and Francesco Scarello. A comparison of structural csp decomposition methods. *Ijcai-99*, 1999.

[17] Nicola Leone Georg Gottlob and Francesco Scarello. A comparison of structural csp decomposition methods. *Artificial Intelligence*, pages 243–282, 2000.

[18] H. Hasfsteinsson H.L. Bodlaender, J. R. Gilbert and T. Kloks. Approximating treewidth, pathwidth and minimum elimination tree-height. In *Technical report RUU-CS-91-1, Utrecht University*, 1991.

[19] J. Larrosa K. Kask, R. Dechter and A. Dechter. Unifying tree-decompositions for reasoning in graphical models. *Artificial Intelligence*, 124.

[20] Vibhav Gogate Kalev Kask, Rina Dechter. Counting-based look-ahead schemes for constraint satisfaction. In *Constraint Programming (CP04)*, pages 317–331, 2004.

[21] K. Kask. Approximation algorithms for graphical models. Technical report, Ph.D. thesis, Information and Computer Science, University of California, Irvine, California, 2001.

[22] J Larrosa and R. Dechter. Dynamic combination of search and variable-elimination in csp and max-csp. *Submitted*, 2001.

[23] S.L. Lauritzen and D.J. Spiegelhalter. Local computation with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B*, 50(2):157–224, 1988.

[24] P. Jeavons M. Gyssens and D. Cohean. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89, 1994.

[25] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[26] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

[27] D. Maier. The theory of relational databases. In *Computer Science Press, Rockville, MD*, 1983.

[28] I. Rish and R. Dechter. Resolution vs. search; two strategies for sat. *Journal of Automated Reasoning*, 24(1/2):225–275, 2000.

[29] D. G. Corneil S. A. Arnborg and A. Proskourowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics.*, 8:277–284, 1987.

[30] R. Seidel. A new method for solving constraint satisfaction problems. In *International Joint Conference on Artificial Intelligece (Ijcai-81)*, pages 338–342, 1981.

[31] P. P. Shenoy. Binary join trees. pages 492–499, 1996.

[32] K. Shoiket and D. Geiger. A proctical algorithm for finding optimal triangulations. In *Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 185–190, 1997.

[33] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM Journal of Computation.*, 13(3):566–579, 1984.