TECHNICAL REPORT
R-92
April 1989

RESEARCH NOTE

Tree Clustering for Constraint Networks*

Rina Dechter and Judea Pearl
*4731 Boelter Hall, Cognitive System Laboratory,
Computer Science Department, University of California,
Los Angeles, CA 90024, U.S.A.*

ABSTRACT

The paper offers a systematic way of regrouping constraints into hierarchical structures capable of supporting search without backtracking. The method involves the formation and preprocessing of an acyclic database that permits a large variety of queries and local perturbations to be processed swiftly, either by sequential backtrack-free procedures, or by distributed constraint propagation processes.

1. Introduction

Solving constraint satisfaction problems (CSPs) usually involves two phases: a preprocessing phase that establishes local consistencies, followed by a backtracking procedure that actually produces the solution desired. While the preprocessing phase normally is accomplished by local, constraint propagation mechanisms, the answer producing phase occasionally runs into difficulties due to excessive backtracking. If a given set of constraints is to be maintained over a long stream of queries, it may be advisable to invest more effort and memory space in restructuring the problem so as to facilitate more efficient answer producing routines. This paper proposes such a restructuring technique, based on clique-tree clustering. The technique guarantees that a large variety of queries could be answered swiftly either by sequential backtrack-free procedures, or by distributed constraint propagation methods.

The technique proposed exploits the fact that the tractability of CSPs is intimately connected to the topological structure of their underlying constraint

* This work was supported in part by the National Science Foundation, grant #DCR 85-01234 and by the Airforce Office of Scientific Research grant #AFOSR-88-0177.

0004-3702/89/$3.50 © 1989, Elsevier Science Publishers B.V. (North-Holland)

graphs (Freuder [14], Dechter [7]). The simplest result in this regard asserts that if the constraint graph is a tree then the corresponding CSP can be solved efficiently, in $O(nk^2)$ steps, where n is the number of variables and k is the number of values. This result is also applicable to processing CSPs of arbitrary topologies; approximations based on trees can be used as heuristics to guide choices in backtracking (Dechter [7]) and tree-solving algorithms can be invoked when subproblems are recognized to be tree-structured (Dechter [8]).

Another important feature of tree topology lies in facilitating unsupervised, constraint propagation mechanisms, often called "relaxation". Distributed relaxation algorithms applied to constraint trees reach equilibrium in time proportional to the tree's diameter and, more significantly, the local consistencies established by such algorithms also guarantee global consistency, namely, any choice of value to any variable in the resultant network is guaranteed to be extendable to a full solution of the CSP. Consequently, the desired solutions can be assembled incrementally, using either backtrack-free search, or a parallel propagation algorithm initiated at some designated center. In networks of a general structure, relaxation algorithms do not yield a globally consistent network, hence, there is no guarantee that a final solution can tractably be assembled thereafter.

A general strategy of utilizing the merits of tree topologies in non-tree CSPs is to form clusters of variables such that the interactions between the clusters are tree-structured, then solve the problem by efficient tree algorithms. This amounts to first, deciding which variables should be grouped together, finding the internally consistent values in each cluster and, finally, processing these sets of values as singleton variables in a tree. Clustering ideas were implemented in specialized constrained-based languages. The notions of "multiple views" in CONSTRAINTS (Sussman [24]) and that of "merging" in THINGLAB (Borning [5]) can be viewed as variants of a clustering strategy.

In this paper we present a general and systematic method of accomplishing this strategy, applicable to both binary and nonbinary CSPs. The method is based on a combination of the theory of *acyclic databases* (Beeri [3]), Freuder's conditions for backtrack-free search [14] and the notion of directional consistency (Dechter [7]). Related methods were also used for structuring statistical databases (Malvestuto [18]), inferences in Bayesian networks (Lauritzen [5], Pearl [21]), and the analysis of belief functions (Shafer [23]).

2. CSPs and Their Graph Representations

A constraint satisfaction problem involves a set of n variables $X_1, \ldots, X_n$, each represented by its domain values, $R_1, \ldots, R_n$ and a set of constraints. A constraint $C_i(X_{i_1}, \ldots, X_{i_j})$ is a subset of the Cartesian product $R_{i_1} \times \cdots \times R_{i_j}$ which specifies which values of the variables are compatible with each other. A solution is an assignment of values to all the variables which satisfy all the constraints and the task is to find one or all solutions. A *binary CSP* is one in

which all the constraints involve only pairs of variables. A binary CSP can be associated with a *constraint graph* in which nodes represent variables and arcs connect pairs of constrained variables. Graph representations for CSPs containing high-order constraints can be constructed in two ways, as a *primal constraint graph* or a *dual constraint graph*. A *primal constraint graph* represents variables by nodes and associates an arc with any two nodes residing in the same constraint. A *dual constraint graph* (called "intersection graph" in database theory (Maier [17]) and being equivalent to a hypergraph representation) represents each constraint by a node (called a *c-variable*) and associates a labeled arc with any two nodes that share variables. The arcs are labeled by the shared variables.

For example, Figs. 1(a) and 1(b) depict the primal and dual constraint graph respectively, of a CSP with variables A, B, C, D, E, F and constraints on the subsets (ABC), (AEF), (CDE), and (ACE) (the constraints themselves are not shown in the figures).

The dual constraint graph representation transforms any nonbinary CSP to a special type of binary CSP: 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). Using this representation and exploiting the structure of the dual constraint graph, we can solve a nonbinary CSP using methods developed for binary CSPs. In particular, if the dual constraint graph is a tree, the problem can be solved in time linear in the number of c-variables, using the tree algorithm described by Dechter [7].

Since trees are desirable structures, we want to transform any constraint graph into a tree. One way of doing it is to form larger clusters of c-variables, another is to identify and remove *redundant arcs*. A constraint is considered redundant if its elimination from the problem does not change the set of solutions [6]. Since all constraints in the dual graph are equalities, an arc can be deleted if its variables are shared by every arc along an alternative path between the two end points. The subgraph resulting from the removal of redundant arcs is called a *join graph*, and it has the following property: for

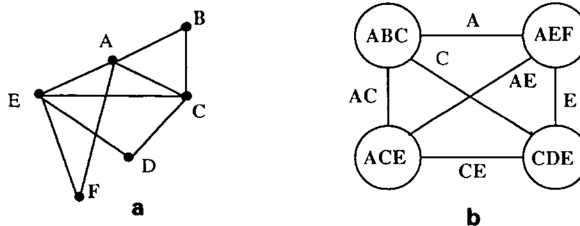

Fig. 1. Primal and dual constraint graphs of a CSP.

each two nodes that share a variable there is at least one path of labeled arcs, each containing the shared variable. A join graph is an equivalent representation to the original dual graph though it may contain fewer arcs.

For example, in Fig. 1(b), the arc between (AEF) and (ABC) can be eliminated because the variable A is common along the cycle (AEF)—A—(ABC)—AC—(ACE)—AE—(AEF) and, so, a consistent assignment to A is ensured by the remaining arcs. By a similar argument we can remove the arcs labeled C and E, thus turning the join graph into a tree, called *join tree*.

A CSP organized as a join tree can be solved efficiently. If there are p constraints in the join tree, each with at most l subtuples, then, a straight application of the algorithm developed for a tree of singletons (i.e., $O(nk^2)$) would yield a solution in $O(pl^2)$. However, by ordering the tuples of each constraint lexicographically, the task of matching two tuples can be reduced to $O(\log l)$ steps (instead of $O(k^2)$ for binary constraints), thus reducing the overall complexity to $O(p \cdot l \cdot \log l)$. CSPs that possess a join tree are called *acyclic databases* (called *acyclic CSPs* here), and their desirable properties are discussed at length by Beeri [3]. Efficient procedures for identifying a join tree of an acyclic database are described by Maier [17].

3. The Tree-Clustering Scheme

Our aim is to transform any CSP into an acyclic representation, even when the dual constraint graph of the original representation of the problem cannot be reduced to a join tree. We do it by systematically forming larger clusters than those given in the dual constraint graphs.

A CSP is acyclic iff its primal graph is both chordal and conformal [3]. A graph G is *chordal* if every cycle of length at least four has a chord, i.e., an edge joining two nonconsecutive vertices along the cycle. A primal graph is *conformal* if each of its maximal cliques corresponds to a constraint in the original CSP.

The clustering scheme is based on an efficient triangulation algorithm (Tarjan [25]) which transforms any graph into a chordal graph by adding edges to it. The maximal cliques of the resulting chordal graph are the clusters necessary for forming an acyclic CSP.

The triangulation algorithm consists of two steps:

Step 1. Compute an ordering for the nodes, using a *maximum cardinality* search.

Step 2. Fill in edges (recursively) between any two nonadjacent nodes that are connected via nodes higher up in the ordering.

The *maximum cardinality search* numbers vertices from 1 to n, in increasing[1]

[1] The order here is the reverse of that used by Tarjan et al. and was changed to simplify the presentation. Such ordering will be called *m-ordering*.

order, always assigning the next number to the vertex having the largest set of previously numbered neighbors (breaking ties arbitrarily). Such ordering will be called *m-ordering*. If no edges are added in Step 2 the original graph is chordal, otherwise the new filled graph is chordal. Tarjan et al. give a maximum cardinality search algorithm that can be implemented in $O(n + \deg)$, where n is the number of variables and deg is the maximum degree. The fill-in step of the algorithm runs in $O(n + m')$ where m' is the number of arcs in the resultant graph. There is no guarantee that the number of edges added by this process is minimal, however, since for chordal graphs the m-ordering requires no fill-in, the fill-in required for nonchordal graphs is usually close to optimal. The preceding discussion suggests the following clustering procedure for CSPs:

Tree-Clustering (T-C)
Begin
(1) Given a CSP and its primal graph, use the triangulation algorithm to generate a chordal primal graph (if the primal graph is chordal no arc will be added).
(2) Identify all the maximal cliques in the primal chordal graph. Let $C_1, \ldots, C_t$ be all such cliques indexed by the rank of their highest nodes.
(3) Form the dual graph corresponding to the new clusters and identify one of its join trees by connecting each C_i to an ancestor C_j $(j < i)$ with whom it shares the largest set of variables [17].
(4) Solve the subproblems defined by the clusters $C_1, \ldots, C_t$ (this amounts to generating a higher-arity constraint from the lower-arity constraints internal to each cluster, i.e., listing the consistent subtuples for the variables in each cluster).
(5) Solve the tree problem with treating the clusters as singleton variables.
    (a) Perform directional arc-consistency (DAC) on the join tree [7].
    (b) Solve the join tree in a backtrack-free manner.
End.

For example, consider a CSP on variables $\{A, B, C, D, E\}$, defined by the constraints: (A, C), (A, D), (B, D), (C, E), (D, E). The primal graph is given in Fig. 2(a). The ordering $d = E, D, C, A, B$ is one possible m-ordering (Fig. 2(b)). The fill-in required by this ordering adds the arc (C, D) and results in the chordal graph of Fig. 2(c). The maximal cliques associated with this graph are: (A, D, C), (D, C, E), and (D, B) (Fig. 2(c)). The dual graph associated with these constraints and one of its associated join trees are shown in Fig. 3(a) and Fig. 3(b) respectively. To solve the problem shown in Fig. 3(b), we first

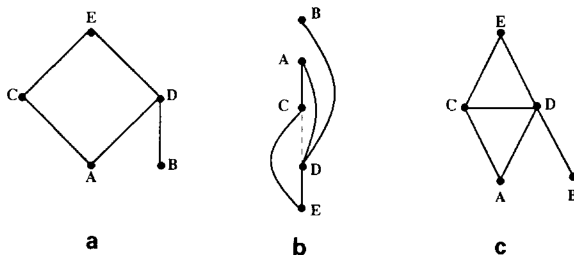

Fig. 2.

solve the three subproblems associated with the sets of variables (A, D, C), (D, C, E) and (D, B), then, using these local solutions as domains for the c-variables, the tree is solved in the usual manner (step (5)). For example, solving subproblem (A, D, C) means finding all assignments to A, D, C which are consistent with the input constraints (A, C) and (A, D).

We can estimate the running time of the algorithm as follows. Given a CSP having n variables, its primal graph may be of $O(n^2)$ of the original problem size (since any two nodes may be connected). Triangulating the primal graph is also bounded by $O(n^2)$ since both the fill-in and the maximal cardinality search are bounded by the size of the resultant graph (number of edges + number of vertices). For the second step (i.e. identifying all maximal cliques) observe that in the filled-in graph, any vertex V and its parent set $C(V)$ (those which are connected to it and precede it in the m-ordering) form a clique. The reason being that any two parent vertices which were not connected in the original graph will be "filled" by the fill-in step of the chordality algorithm. Therefore, to enumerate all maximal cliques we can determine the cliques $C(V)$ in decreasing order of V, discarding a newly generated clique that is contained in a previous clique. In Fig. 2(b), the clique (BD), associated with B, is identified first, next the clique (ACD) (indexed by A), then (CDE). Since the last clique, (DE), is contained in (CDE) it is discarded. In this process each arc will be tested once to determine adjacency and, therefore, the complexity of this step is $O(|E'|)$ when E' is the set of edges in the filled graph. Notice that the maximum number of cliques is n.

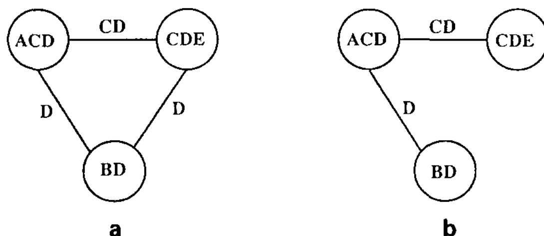

Fig. 3.

The third step, determining the join tree, is linear in the size of the triangulated primal graph. The fourth step requires solving the subproblems defined by each clique. If r is the size of the largest clique and k is the number of values for each variable, this step is $O(k^r)$ and may dominate the overall computation. Finally, the last step of solving the join tree is $O(n \cdot t \log t)$ where t is the maximum number of solutions in each clique. This step is performed by executing directional arc-consistency (in the specialized manner mentioned earlier) from leaves to root [7] (step (5a)), and then finding a solution in a backtrack-free manner (step (5b)). Thus, the overall complexity of the clustering scheme is bounded by:

$$O(n^2) + O(k^r) + O(n \cdot t \log t) . \quad (1)$$

Since $t \leq k^r$, the third summand is bounded by $O(n \cdot k^r \cdot r \log k)$, which yields the bound

$$O(n^2) + O(k^r) + O(n \cdot k^r \cdot r \log k) = O(k^r) = O(nr \cdot k^r) . \quad (2)$$

The space complexity is $O(n \cdot k^r)$ since there are at most n cliques whose explicated solution set may be exponential in the size of the clique.

The question is whether some computation can be saved in steps (4) and (5), by executing the clustering steps in a coordinated way. For example, it appears wasteful to independently solve two adjacent cliques, only to find out later that many of the solutions found are incompatible with each other. A more economical way would be to consult the solutions found in one clique for pruning the set of solutions assembled in adjacent cliques. Such possibilities are offered by enforcing local *consistency* as shown in the next subsection.

4. Adaptive-Consistency

Freuder [14] has studied the level of local consistency required to guarantee that solutions can be retrieved in a "backtrack-free" manner. We show how this theory, coupled with the notion of directional consistency (Dechter [7]), leads to a clustering scheme similar to that of Section 3.

The *width of a node* in an ordered graph is the number of links connecting it to nodes lower in the ordering. The *width of an ordering* is the maximum width of nodes in that ordering, and the *width of a graph* is the minimal width of all its orderings.

A CSP is *i-consistent* if for any set of $i-1$ variables along with values for each that satisfy all the constraints among them, there exists a value for any ith variable, such that the i values together satisfy all the constraints among the i variables. *Strong i-consistency* holds when the problem is *j-consistent* for $j \leq i$. Given an ordering d, *directional i-consistency* (d-i-consistency for short) re-

quires only that any consistent instantiation of $i-1$ variables can be consistently extended by any variable that succeed all of them in the ordering d. *Strong d-i-consistency* can be defined accordingly. The following theorem summarizes the conditions for backtrack-free search:

Theorem (Freuder [14], Dechter [7]). *An ordered constraint graph is backtrack-free if the level of directional strong consistency along this order is greater than the width of the ordered graph.*

When a problem is not i-consistent, algorithms enforcing i-consistency can be applied to it (Freuder [13]), e.g., the algorithms known as *arc-consistency* and *path-consistency* enforce 2-consistency and 3-consistency respectively (Montanari [20], Mackworth [16], Dechter [7], Mohr [19]). It may seem that the above theorem can be used as follows. Given a CSP, find the width of its graph (Freuder presents a linear-time algorithm for finding the width, W, of a graph), perform a $(W+1)$-consistency algorithm, then solve the problem in a backtrack-free manner. Unfortunately, achieving i-consistency $(i > 2)$ often requires the addition of constraints which amounts to adding arcs to the constraint graph and increasing its width, thus violating the conditions for backtrack-free search. The following procedure, first presented in [7] takes this issue into consideration. A similar algorithm, suggested by Seidel [22] embodies, essentially, the same idea.

Given an ordering, d, we establish *d-i-consistency* recursively, letting i change dynamically from node to node to match its width at the time of processing. Nodes are processed in decreasing order, so that by the time a node is processed, its final width is determined and the required level of consistency can be achieved. For each variable, X, let PARENTS(X) be the set of all variables connected to it and preceding it in the graph. The parents of each variable are computed only when it needs to be processed.

Adaptive-Consistency($X_1, \ldots, X_n$)
Begin
(1) for $i = n$ to 1 by -1 do steps (2)–(4)
(2) compute PARENTS(X_i)
(3) connect all elements in PARENTS(X_i) (if they are not yet connected)
(4) perform consistency(X_i, PARENTS(X_i))
(5) find a solution using backtrack in the ordering ($X_1, \ldots, X_n$)
End

The procedure consistency(V, SET) generates and records those tuples of variables in SET that can be consistent with at least one value of V. The procedure may impose new constraints over clusters of variables as well as

tighten existing constraints. Note that the procedure can generate constraints that contain other constraints. When the procedure terminates, backtrack can solve the problem, in the order prescribed without encountering any dead end. The topology of the *induced graph* (identical to the one generated by directional path-consistency) can be found prior to executing the procedure, by recursively connecting any two parents sharing a common successor.

Consider our example of Fig. 2 in an ordering (E, D, C, A, B) shown in Fig. 4(a). The Adaptive-Consistency algorithm proceeds from B to E and imposes consistency constraints on the parents of each processed variable. B is chosen first and the algorithm enforces a 2-consistency on D (namely an arc-consistency on (D, B)), since the width of B is 1. A is selected next and, having width 2, the algorithm enforces a 3-consistency on its parents $\{C, D\}$. This operation may require that a constraint between C and D be added, and in that case an arc (C, D) is added. When the algorithm reaches node C its width is 2 and, therefore, a 3-consistency is enforced on C's parents $\{E, D\}$. The arc (E, D) already exists so this operation may merely tighten the corresponding constraint. The resulting graph is given in Fig. 4(b), and its dual constraint graph consisting of all recorded constraints, is shown in Fig. 4(c).

Let $W(d)$ be the width of the ordering d and $W^*(d)$ the width of the induced graph. The complexity of solving a problem using the Adaptive-Consistency preprocessing phase (steps (1)–(4)) and then backtracking (freely) along the order d (step (5)) is dominated by the former. The worst-case complexity of the "consistency(V, PARENT(V)) step" is exponential in the cardinality of variable V and its parents. Since the maximal size of the parent sets is equal to the width of the induced graph we see that solving the CSP along the ordering d is $O(n \cdot \exp(W^*(d) + 1))$.

5. Relationships between Adaptive-Consistency (A-C) and Tree-Clustering (T-C)

The two schemes presented, although unrelated at first glance, share many interesting features. First, for any given ordering d, the set of fill-in arcs added by triangulation, is equal to the set of arcs added by the Adaptive-Consistency

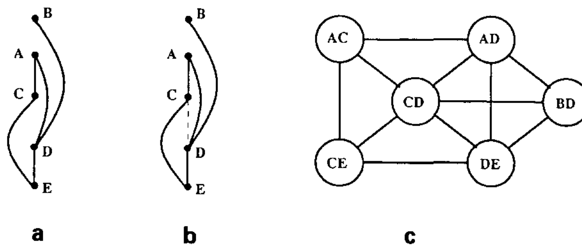

Fig. 4.

scheme. Both methods recursively connect sets of nodes that share a common successor in the ordering, so the two will induce the same final graph if initiated on the same ordered graph (see Figs. 2(b) and 4(b)). In particular, the induced graph is always chordal and, if the original graph is chordal and ordered by a maximum cardinality search, its width will not change (no arcs are added in this case).

In addition, a strong structural resemblance exists between the clusters chosen by T-C and the constraints (new or old) recorded by A-C. In each maximal clique C of size r (in the induced graph) A-C will record or tighten at least one constraint of size $r-1$. If C contains another clique C' of size r' then this, too, is associated with A-C recording one constraint of size $r'-1$. Namely, every cluster in T-C (i.e., a maximal clique) is represented in A-C by the constraints originally contained in that cluster (some of which may be tightened), and at most one additional constraint for each size less than the cluster's cardinality. In Figs. 5(a) and 5(b), we present once again the clusters generated by T-C and the constraints recorded by A-C.

Rough asymptotic bounds on the time and space complexity of both schemes reveal that they are about the same. If $W^*(d)$ is the width of the induced graph, then $W^*(d)+1$ is the size of the largest clique and, therefore, both A-C and T-C are space-bounded by $O(n \cdot k^{W^*(d)})$ and time-bounded by $O(n \cdot k^{W^*(d)})$ and $O(nr \cdot k^{W^*(d)})$ respectively, k being the number of values. In practice, however, we may find cases favoring either one of the two schemes space-wise, because the explicit representation of T-C may sometimes be more economical.

Asymptotically, the worse-case bounds of both schemes, dominated by the exponent of both expressions, is roughly the same, and the magnitude of these exponents is determined by the ordering, d, used (note that T-C is not compelled to use the maximum cardinality ordering which only provides a useful ordering heuristic applicable for both schemes). The algorithms' bound can be further tightened to yield $O(\exp(W^*+1))$ where $W^* = \min_d \{W^*(d)\}$. However, computing an optimal d was shown to be an NP-complete task (Arnborg [1]), and among the various heuristic orderings studied in the

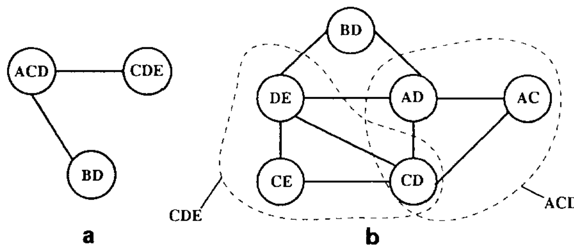

Fig. 5.

literature (Bertelé [4]), the most popular are the minimal width and the m-orderings. The ease of finding these orderings enables us to calculate $W^*(d)$ under both orderings, and take the lowest value as a better upper bound estimate of W^*. Moreover, any minimum width ordering, denoted d_{mw}, can be used for generating both a lower and an upper bound for W^* since

$$W(d_{mw}) \leq W^* \leq W^*(d_{mw}) .$$

In terms of actual time complexity we argue that A-C outperforms T-C, and in effect can be considered an efficient approach to tree clustering. The reason is that clusters are not assembled independently, but are pruned during construction. Consider the binary CSP presented by the constraint graph of Fig. 6(a); the graph is chordal and doesn't change by either scheme. Assume the m-ordering of Fig. 6(b) and the join tree of Fig. 6(c). When T-C solves the problem, the two subproblems (ABC) and (BCD) must first be solved *independently*. (ABC) may be solved by computing the constraint that A induces on (BC), then listing all consistent triplets. The same can be done for (BCD) but, since the two subproblems are solved independently, the solution found for (BCD) conforms only to the original constraint on (BC) not the one tightened by solving subproblem (ABC). Thus, several triplets in (BCD) would be generated and listed redundantly, only to be discarded, once the solutions of the two subproblems interact via the directional arc-consistency of step (5a).

Adaptive-Consistency eliminates this redundancy. Proceeding along the order D, C, B, A, variable A tightens the constraint on (B, C), then B tightens the constraint on (C, D) and, finally, C induces a unary constraint on D. The problem can now be solved in a backtrack-free manner along the original ordering. Thus, A-C constructs, in effect, a join tree that is already directional arc-consistent and, so, renders step (5a) of T-C unnecessary. The only difference between the join tree produced by A-C and that resulting from step (4) of T-C is that A-C does not explicitly enumerate the domains of the c-variables but, instead, represents them as conjunctions of lower-arity constraints. If

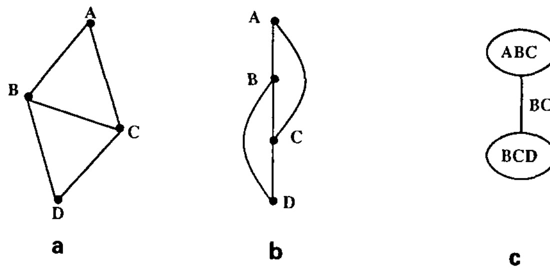

Fig. 6.

desired, these domains can easily be enumerated using backtrack search (step (5) of A-C) and, due to the additional constraints induced by A-C, this search is backtrack-free. Such enumeration will result in a join tree that is fully arc-consistent, because the backtrack search ensures that each solution found for a c-variable has a matching solution at its parent c-variable.

The question arises whether there is ever a need to fully explicate the domain of each clique in the join tree. Obviously, if the ultimate task is merely finding one (or all) solution to the given CSP, then the representation constructed by the A-C algorithm is sufficient; solutions can be produced without backtracking in the ordering prescribed by A-C. However, not all applications are suitably handled by processes committed to a fixed ordering. For example, consider the query: "Is there a solution in which variable X_j attains the value x?". The representation generated by T-C enables us to answer this query by just scanning the domain of one clique that contains the variable X_j. The representation generated by A-C, on the other hand, requires that a global search be initiated from the first variable (step (5) of A-C), and this will often require explicating the domains of more than one clique. In general, if the ultimate task is to maintain an effective database for answering a variety of queries, a balanced, undirectional representation is preferred. Since Tree-Clustering provides a more explicit representation, it will facilitate more efficient query-answering routines, at the expense of the additional space required for storing the explicit domains.

6. Conclusions

Tree clustering offers a systematic way of regrouping elements into hierarchical structures capable of supporting information retrieval without backtracking. The basic Tree-Clustering scheme involves triangulating the constraint graph, identifying the maximal cliques of the triangulated graph, solving the constraints associated with each clique and organizing the solutions obtained in a tree structure. A routine called Adaptive-Consistency has been identified as an effective method of assembling the desired tree.

Once the clusters are formed and their join tree established and processed, the resulting structure offers an effective database, to be amortized over many problem instances. A large variety of queries could be answered swiftly either by sequential backtrack-free procedures, or by distributed constraint propagation processes. A typical application for this strategy is the management of dynamically changing environments, where it is often required to make new assumptions so as to keep the knowledge base consistent with incoming observations. A common task under such conditions is to determine whether a specific proposition $X = x$ is entailed by the constraint network and a set of default assumptions, modelled as temporary instantiations of a select set of variables. Truth maintenance systems (also known as reason maintenance

systems and belief maintenance systems) which were developed to handle such tasks (Doyle [12], de Kleer [11]) compromise completeness for efficiency. In other words, they confirm only those propositions that can be proven by efficient constraint propagation algorithms, leaving others unconfirmed. Tree clustering methods should enable us to bridge this completeness gap by first organizing the knowledge into a join tree, then verifying entailment in constraint propagation fashion (Dechter [9]).

Clearly, this method is useful only when environmental conditions undergo local changes, i.e., those that do not alter the structure of the join tree. These include changes in the domains of individual variables or adjustments of constraints which reside within a single clique. Such changes are indeed the ones encountered in reasoning about physical or biological systems; the majority of changes result from observations made on the states of individual variables, while the bulk of knowledge remains intact.

The Tree-Clustering scheme can facilitate efficient computation of many functions which are easily solvable on a tree of binary relationships. Such applications include belief propagation in Bayesian networks (Lauritzen [15], Pearl [21]), belief functions computations (Shafer [23]) and constraint optimization (Dechter [10]). A recent paper (Arnborg [2]) describes a language which states properties and problems which are easy for tree-decomposable graphs. Future experimental work is required to compare Tree-Clustering and backtrack algorithms in order to determine when the advantages of these schemes (as manifested by their worse-case bounds) are translated into an actual improvement in performance.

ACKNOWLEDGMENT

We thank Johan de Kleer for reading and commenting on this paper.

REFERENCES

1. Arnborg, S., Corneil, D.G. and Proskurowski, A., Complexity of finding embeddings in a *k*-tree, *SIAM J. Algebraic Discrete Methods* **8** (2) (1987) 277–284.
2. Arnborg, S., Lagergren, J. and Seese, D., Problems easy for tree-decomposable graphs, in: *Proceedings 15th International Colloquium on Automata Languages and Programming*, Tampere, Finland (1988).
3. Beeri, C., Fagin, R., Maier, D. and Yannakakis, M., On the desirability of acyclic database schemes, *J. ACM* **30** (3) (1983) 479–513.
4. Bertelé, U. and Brioschi, F., *Nonserial Dynamic Programming* (Academic Press, New York, 1972).
5. Borning, A., The programming language aspects of Thinglab, a constraint-oriented simulation laboratory, *ACM Trans. Program. Lang. Syst.* **3** (4) (1981) 353–387.
6. Dechter, A. and Dechter, R., Removing redundancies in constraint networks, in: *Proceedings AAAI-87*, Seattle, WA (1987) 105–109.
7. Dechter, R. and Pearl, J., Network-based heuristics for constraint-satisfaction problems, *Artificial Intelligence* **34** (1) (1988) 1–38.

8. Dechter, R. and Pearl, J., The cycle-cutset method for improving search performance in AI applications, in: *Proceedings 3rd IEEE Conference on AI Applications*, Orlando, FL (1987) 224–230.
9. Dechter, R. and Dechter, A., Belief maintenance in dynamic constraint networks, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 37–42.
10. Dechter, R., Dechter, A. and Pearl, J., Optimization in constraint-networks, in: *Proceedings Berkeley Conference on Influence Diagrams* (Wiley, New York, 1988).
11. de Kleer, J., An assumption-based TMS, *Artificial Intelligence* **28** (1986) 127–162.
12. Doyle, J., A truth maintenance system, *Artificial Intelligence* **12** (3) (1979) 231–272.
13. Freuder, E.C., Synthesizing constraint expressions, *Commun. ACM* **21** (11) (1978) 958–965.
14. Freuder, E.C., A sufficient condition of backtrack-free search, *J. ACM* **29** (1) (1W982) 24–32.
15. Lauritzen, S.L. and Spiegelhalter, D.J., Local computations with probabilities on graphical structures and their applications to expert systems, *J. R. Stat. Soc. B.* **50** (1988) 127–224.
16. Mackworth, A.K. and Freuder, E.C., The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artificial Intelligence* **25** (1) (1985) 65–74.
17. Maier, D., *The Theory of Relational Databases* (Computer Science Press, Rockville, MD, 1983).
18. Malvestuto, F.M., Answering queries in categorical databases, in: *Proceedings Sixth Conference on the Principles of Database Systems*, San Diego, CA (1987) 87–96.
19. Mohr, R. and Henderson, T.C., Arc and path consistency revisited, *Artificial Intelligence* **28** (2) (1986) 225–233.
20. Montanari, U., Networks of constraints: Fundamental properties and applications to picture processing, *Inf. Sci.* **7** (1974) 95–132.
21. Pearl, J., *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* (Morgan Kaufmann, Los Altos, CA, 1988).
22. Seidel, R., A new method for solving constraint-satisfaction problems, in: *Proceedings IJCAI-81*, Vancouver, BC (1981) 338–342.
23. Shafer G., and Shenoy, P.P., Bayesian and belief-function propagation, Tech. Rept. #192 University of Kansas School of Business, Lawrence, KS (1988).
24. Sussman, G.J. and Steele Jr, G.L., CONSTRAINTS: A language for expressing almost-hierarchical descriptions, *Artificial Intelligence* **14** (1) (1980) 1–39.
25. Tarjan, R.E. and Yannakakis, M., Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs, *SIAM J. Comput.* **13** (3) (1984) 566–579.

Received August 1987; revised version received October 1988