Hierarchical arc consistency: exploiting structured domains in constraint satisfaction problems

ALAN K. MACKWORTH, JAN A. MULDER,[1] AND WILLIAM S. HAVENS
Laboratory for Computational Vision, Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada V6T 1W5

Received July 19, 1985
Revision accepted December 19, 1985

Constraint satisfaction problems can be solved by network consistency algorithms that eliminate local inconsistencies before constructing global solutions. We describe a new algorithm that is useful when the variable domains can be structured hierarchically into recursive subsets with common properties and common relationships to subsets of the domain values for related variables. The algorithm, HAC, uses a technique known as hierarchical arc consistency. Its performance is analyzed theoretically and the conditions under which it is an improvement are outlined. The use of HAC in a program for understanding sketch maps, Mapsee3, is briefly discussed and experimental results consistent with the theory are reported.

Key words: constraint satisfaction problems, network consistency algorithms, arc consistency, map understanding, computational vision.

Les problèmes de satisfaction de contrainte peuvent être résolus par des algorithmes de consistance de réseau qui éliminent les inconsistances locales avant de construire des solutions globales. Nous décrivons un nouvel algorithme qui s'avère utile lorsque les domaines variables peuvent être structurés hiérarchiquement en sous-ensembles récursifs possédant des propriétés communes et des relations communes avec les sous-ensembles des valeurs de domaine pour les variables reliées. L'algorithme, HAC, utilise une technique connue sous le nom de consistance d'arc hiérarchique. Son fonctionnement est analysé d'un point de vue théorique et les améliorations qu'il peut apporter sont mises en évidence. L'utilisation de HAC dans un programme de compréhension d'esquisses de cartes, Mapsee3, est brièvement discutée et des résultats expérimentaux consistants avec la théorie sont mentionnés.

Mots clés: problèmes de satisfaction de contrainte, algorithme de consistance de réseau, consistance d'arc, compréhension de carte, vision automatique.

[Traduit par la revue]

Comput. Intell. 1, 118–126 (1985)

1. Introduction

In this paper we show how to exploit aspects of the intrinsic structure of variable domains when using a network consistency algorithm to solve a constraint satisfaction problem. A prerequisite brief review of the basic concepts is presented first. For a fuller explanation the reader should consult the original material (Waltz 1972; Montanari 1974; Mackworth 1977*a*; Freuder 1978; Haralick and Elliott 1980; Freuder 1982; Mackworth and Freuder 1985).

1.1. Constraint satisfaction and network consistency

A constraint satisfaction problem (CSP) is defined as follows: Given a set of n variables each with an associated domain and a set of constraining relations each involving a subset of the variables, find all possible n-tuples such that each n-tuple is an instantiation of the n variables satisfying the relations. In this paper we shall only consider CSP's in which the relations are unary and binary. This restriction is not necessary for consistency techniques to be applied (Mackworth 1977*b*; Freuder 1978).

Since graph colouring is an NP-complete CSP, it is most unlikely that a polynomial time algorithm exists for solving general CSP's. Accordingly, the class of network consistency algorithms was invented. These algorithms do not necessarily solve a CSP completely but they eliminate, once and for all, local inconsistencies that cannot participate in any global solutions. These inconsistencies would otherwise have been repeatedly discovered by any backtracking solution. One role for network consistency algorithms is as a preprocessor for subsequent backtrack search, or they can be interspersed with case analysis or simple domain splitting to recover the complete set of solutions to the CSP. A k-consistency algorithm removes all inconsistencies involving all subsets of size k of the n variables. For example, the node, arc, and path consistency algorithms detect and eliminate inconsistencies involving $k = 1, 2$, and 3 variables, respectively. Freuder's (1978) generalization of those algorithms for $k = 1,\ldots,n$ can be used to produce the complete set of solutions to the CSP.

1.2. Node and arc consistency

The algorithms below are reprinted from a previous paper (Mackworth 1977*a*) that should be consulted for a full explanation. The domain of variable x_i is D_i, P_i is the unary predicate on x_i, and P_{ij} is the binary constraint predicate on the variables x_i and x_j corresponding to an edge between vertices v_i and v_j in the constraint graph G. The edge between i and j is replaced by the directed arc from i to j and the arc from j to i as they are treated separately by the algorithms. Let the number of variables be n, the number of binary constraints be e (the number of edges in the constraint graph), and the edge degree of v_i be d_i. The time unit used for our complexity measures is the application of a unary or binary predicate. To simplify the description of the complexity results, in this section we assume that each D_i is the same initial size a.

The node consistency algorithm NC-1 simply ensures that all values in D_i satisfy P_i by removing those that do not.

```
procedure NC (i):
D_i ← D_i ∩ {x | P_i (x)}
begin
  for i ← 1 until n do NC(i)
end
```

NC-1: the node consistency algorithm

An arc consistency algorithm is a symbolic relaxation

[1]Current address: Department of Mathematics, Statistics and Computer Science, Dalhousie University, Halifax, N.S., Canada B3H 4H8.

algorithm that establishes the strong arc consistency condition on each arc of G. The arc (i, j) from v_i to v_j is strongly arc consistent iff:

(1) v_i is node consistent and

(2) for each value in D_i there is at least one value in D_j that is compatible with it (such that P_{ij} is satisfied).

The algorithm AC-3 (Mackworth 1977*a*) is an efficient arc consistency algorithm:

```
procedure REVISE ((i,j)):
begin
  DELETE ← false
  for each x ∈ D_i do
      if there is no y ∈ D_j such that P_ij(x,y) then
        begin
          delete x from D_i
          DELETE ← true
        end;
  return DELETE
end

1  begin
2    for i ← 1 until n do NC(i)
3    Q ← {(i,j) | (i,j) ∈ arcs (G), i ≠ j}
4    while Q not empty do
5      begin
6        select and delete any arc (k,m) from Q
7        if REVISE ((k,m)) then Q ← Q ∪ {(i,k) | (i,k) ∈ arcs (G),
                                          i ≠ k, i ≠ m}
8      end
9  end
```

AC-3: an arc consistency algorithm

REVISE $((i,j))$ makes arc (i,j) strongly arc consistent. AC-3 applies REVISE to each arc of G in turn. It only reconsiders arc (i,j) if it has potentially become inconsistent again because of a deletion from D_j.

Mackworth and Freuder (1985) showed that the time complexity of AC-3 is at best $\Omega(a^2e)$ and at worst $O(a^3e)$. This somewhat surprising worst case behavior for a serial relaxation algorithm, linear in the number of constraints, confirms the empirical results of using AC-3 in several experimental systems (Waltz 1972; Mackworth 1977*b*; Havens and Mackworth 1983). The time complexity does depend heavily on the domain size a. As more realistic problems are tackled the domain size increases substantially. Accordingly we were motivated to look for ways of coping with larger domain sizes.

2. Intensional domains and predicates

The arc consistency algorithm described above, AC-3, assumes that the domains are supplied extensionally as unstructured sets, listing the finite number of members. Consistency techniques can, however, be applied to CSP's in which the domains do not satisfy that assumption. For example, the domains could be supplied intensionally as descriptions. For any infinite domain this is clearly a necessity. A good example of this is space planning (Mackworth 1977*a*).

In a two-dimensional facility or VLSI layout problem the domains (possible placements of the objects) might be given intensionally as subsets of R^2 by describing their boundaries. The constraints, similarly, would be described as intensional predicates. The only necessary adequacy requirement on domain and constraint representations is that they allow one to carry out the domain restriction operation of REVISE. The version of REVISE used by AC-3 assumes an extensional, unstructured set representation of D_i. A more abstract definition of REVISE that does not make that assumption is as follows:

```
1  procedure REVISE-A((i,j)):
2  begin
3    Δ ← {x | (x ∈Δ_i) ∧ [(∃y)(y ∈Δ_j) ∧ P_ij (x,y)]};
4    DELETE ← (Δ ⊂ Δ_i);
5    if DELETE then Δ_i ← Δ;
6    return DELETE
7  end
```

We are now using Δ_i to represent the dynamic value of the currently permissible domain of variable v_i, which monotonically decreases in size. The set of domains, $\{\Delta_i\}$, will have to be initialized by the following statement to be inserted into AC-3 between steps 1 and 2:

```
1.5  for i ← 1 until n do Δ_i ← D_i
```

and similarly NC(i) becomes

```
procedure NC(i):
Δ_i ← Δ_i ∩ {x | P_i (x)}
```

Line 3 of REVISE-A is the *domain restriction operation*. If the proper subset test of line 4 sets DELETE to **true** then the restricted domain Δ replaces the old value of Δ_i and REVISE-A returns **true** to indicate that a domain restriction has occurred. If AC-3 uses REVISE-A instead of REVISE then it is suitable for this more general class of CSP's, provided, of course, that the domain and predicate representations used by NC and REVISE-A allow the domain restrictions. Incidentally, in the language of relational data base theory (Maier 1983) the domain restriction of REVISE-A is a semi-join. However, in general, relational data base theory makes *the extensional assumption*, both for the domains and the relations.

The abstract algorithm REVISE-A, developed to allow for intensional domains and predicates, will be used as the basis for the efficient treatment of structured extensional domains.

3. Hierarchical domains

An important technique for handling large domains is to exploit their internal structure. Indeed, for many real world problems the domain elements often cluster into sets with common properties and relations. Those sets, in turn, group to form higher level sets. This clustering or categorization into "natural kinds" can be represented as a specialization/generalization (is-a) hierarchy (Havens and Mackworth 1983). The main theme of this paper is the exploitation of the structure provided when the domains can be naturally represented as specialization hierarchies. Each domain can be interpreted as a domain graph with each vertex corresponding to a set of elements and each arc the subset relation between sets. Domain graphs are, of course, quite distinct from the constraint graphs introduced earlier.

In general, a domain graph is not a strict tree since a set may be a direct subset of more than one superset; the general characterization of the resultant "tangled" hierarchy is as a directed acyclic graph representing the lattice induced by the partial ordering of the subset relation. For the purposes of this paper we shall assume that the domain hierarchies are singly rooted strict trees. Each subset has only one direct superset and the subsets of a set are mutually exclusive and exhaustive. Without further loss of generality, we shall assume that the trees are binary: the root represents the entire domain, each non-

singleton set has two subsets, and the leaves are the singleton sets, one for each domain element.

The aim is to make REVISE, the inner loop of AC-3, considerably more efficient by reducing the number of predicate evaluations it must perform. The currently active elements of a domain can be represented by a set of tree vertices that dominate those members. REVISE can then retain, eliminate, or further examine entire subsets of the domain with one or two predicate evaluations.

To achieve this aim, two new families of predicates will be introduced. These new predicates apply not to pairs of individual domain elements for two related variables but to pairs of subsets of the domains of the two variables. These new predicates are easily computed from the predicates supplied with the CSP relating individual domain elements. The expectation is that the domains are structured so that the elements of a subset frequently share consistency properties that permit them to be retained or eliminated as a unit.

For the sake of relative simplicity in the notation, we shall assume that each domain D_i can be structured as a balanced binary tree of depth m, thus all the domains have the same number of elements: $a = 2^m$. We shall use D_i to represent the original domain for variable x_i and Δ_i to represent the active subset of D_i for x_i at any point in the symbolic relaxation process. Δ_i may be implemented as a set of the active subdomains of D_i. The subdomains of D_i are $\{D_i^{ks}\}$ which can be arranged as a tree as shown in Fig. 1, where an arc indicates that the subdomain at the bottom of the arc is a direct subset of the subdomain at the top. The notation for D_i^{ks} indicates that it is on the kth level of the domain tree for D_i and it is the sth subdomain at that level.

D_i^{ks} is partitioned into two mutually exclusive subsets of equal size, $D_i^{(k-1)(2s-1)}$ and $D_i^{(k-1)2s}$. In other words, the following conditions obtain on the subdomains.

For $k = 0,1,2,\ldots, m$ and $s = 1,2,3,\ldots, 2^{m-k}$,

$$D_i^{ks} = D_i^{(k-1)(2s-1)} \cup D_i^{(k-1)2s}$$

$$D_i^{(k-1)(2s-1)} \cap D_i^{(k-1)2s} = \phi$$

and

$$| D_i^{ks} | = 2^k$$

In the algorithm we shall develop, Δ_i, the set of still active elements of the original D_i, is the union of a number of mutually exclusive sets D_i^{ks}. At all times, $\Delta_i \subseteq D_i$.

Suppose

$$\Delta_i = \bigcup_q \Delta_i^q$$

and

$$\Delta_j = \bigcup_r \Delta_j^r$$

We call each Δ_i^q an *abstract label* of Δ_i. Each abstract label is identical to a subset of D_i, D_i^{ks} for some k and s. In the algorithm, Δ_i is represented by the set $\{\Delta_i^q\}$.

We must now efficiently implement the domain restriction step of REVISE-A (line 3 of REVISE-A, repeated here for convenience):

3 $\Delta \leftarrow \{x \mid (x \in \Delta_i) \wedge [(\exists\, y)(y \in \Delta_j) \wedge P_{ij}(x,y)]\}$

Informally, what we wish to do is test each abstract label Δ_i^q of Δ_i, using a generalized version of REVISE. If there is a Δ_j^r such that *every* domain element in Δ_i^q is compatible with *some* element in Δ_j^r, then Δ_i^q survives unchanged in Δ_i. If not, then check to see if there is a Δ_j^r such that *some* element in Δ_i^q is compatible with *some* element in Δ_j^r. If not, then Δ_i^q is simply removed from Δ_i. If there is such a Δ_j^r, then Δ_i^q is replaced in Δ_i by its two child subdomains. When all Δ_i^q subsets of Δ_i have been processed this way (including the new ones generated in the course of processing) then the arc (i,j) is arc consistent.

We generalize the definition of arc consistency as follows. An abstract label pair (Δ_i^q, Δ_j^r) is *strongly hierarchically arc consistent* iff the set of leaf labels below Δ_i^q is strongly arc consistent with the set of leaf labels below Δ_j^r. The arc (i,j) is strongly hierarchically arc consistent iff each abstract label of Δ_i is strongly hierarchically arc consistent with some abstract label of Δ_j.

4. Hierarchical predicates

In order to implement a generalized REVISE we need two new sets of predicates derived from $P_{ij}(x,y)$. These are predicates on the abstract labels D_i^{ks} needed to carry out the operations described above. We define $A_{ij}^{kl}(D_i^{ks}, D_j^{lt})$ to be true iff for *all* elements belonging to D_i^{ks} there is an element of D_j^{lt} compatible with it. A_{ij}^{kl} tests subsets at level k of D_i against subsets at level l of D_j. Analogously, we define $S_{ij}^{kl}(D_i^{ks}, D_j^{lt})$ to be true iff for *some* element belonging to D_i^{ks}, there is an element of D_j^{lt} compatible with it.

$$A_{ij}^{kl}(D_i^{ks}, D_j^{lt}) \text{ iff } (\forall\, x \in D_i^{ks})(\exists\, y \in D_j^{lt}) P_{ij}(x,y)$$

We can compute A_{ij}^{kl} inductively on k and l as follows.

Suppose

$$D_i = \{a_s \mid s = 1, 2, \ldots, 2^m\}$$

and

$$D_j = \{b_t \mid t = 1, 2, \ldots, 2^m\}$$

then let $D_i^{0s} = \{a_s\}$ and $D_j^{0t} = \{b_t\}$.

At the 0th levels of domains D_i and D_j, each subset is a singleton set. Since "all" and "some" are equivalent for a domain size of one, we know that A_{ij}^{00}, S_{ij}^{00}, and P_{ij} are identically equal.

[1] $A_{ij}^{00}(D_i^{0s}, D_j^{0t}) = P_{ij}(a_s, b_t)$

[2] $A_{ij}^{0l}(D_i^{0s}, D_j^{lt}) = A_{ij}^{0,l-1}(D_i^{0s}, D_j^{(l-1)(2l-1)}) \vee A_{ij}^{0,l-1}(D_i^{0s}, D_j^{(l-1)2l})$ for $l = 1,2,\ldots, m$

[3] $A_{ij}^{kl}(D_i^{ks}, D_j^{lt}) = A_{ij}^{(k-1)l}(D_i^{(k-1)(2s-1)}, D_j^{lt}) \wedge A_{ij}^{(k-1)l}(D_i^{(k-1)2s}, D_j^{lt})$ for $k = 1,2,\ldots, m$ and $l = 1,2,\ldots, m$

Suppose, as a simple example, that

$$D_1 = \{a_1, a_2, a_3, a_4\} \qquad D_2 = \{b_1, b_2, b_3, b_4\}$$

then

$$D_1^{01} = \{a_1\} \quad D_1^{02} = \{a_2\} \quad D_1^{03} = \{a_3\} \quad D_1^{04} = \{a_4\}$$
$$D_1^{11} = \{a_1, a_2\} \quad D_1^{12} = \{a_3, a_4\}$$
$$D_1^{21} = \{a_1, a_2, a_3, a_4\}$$

and

$$D_2^{01} = \{b_1\} \quad D_2^{02} = \{b_2\} \quad D_2^{03} = \{b_3\} \quad D_2^{04} = \{b_4\}$$
$$D_2^{11} = \{b_1, b_2\} \quad D_2^{12} = \{b_3, b_4\}$$
$$D_2^{21} = \{b_1, b_2, b_3, b_4\}$$

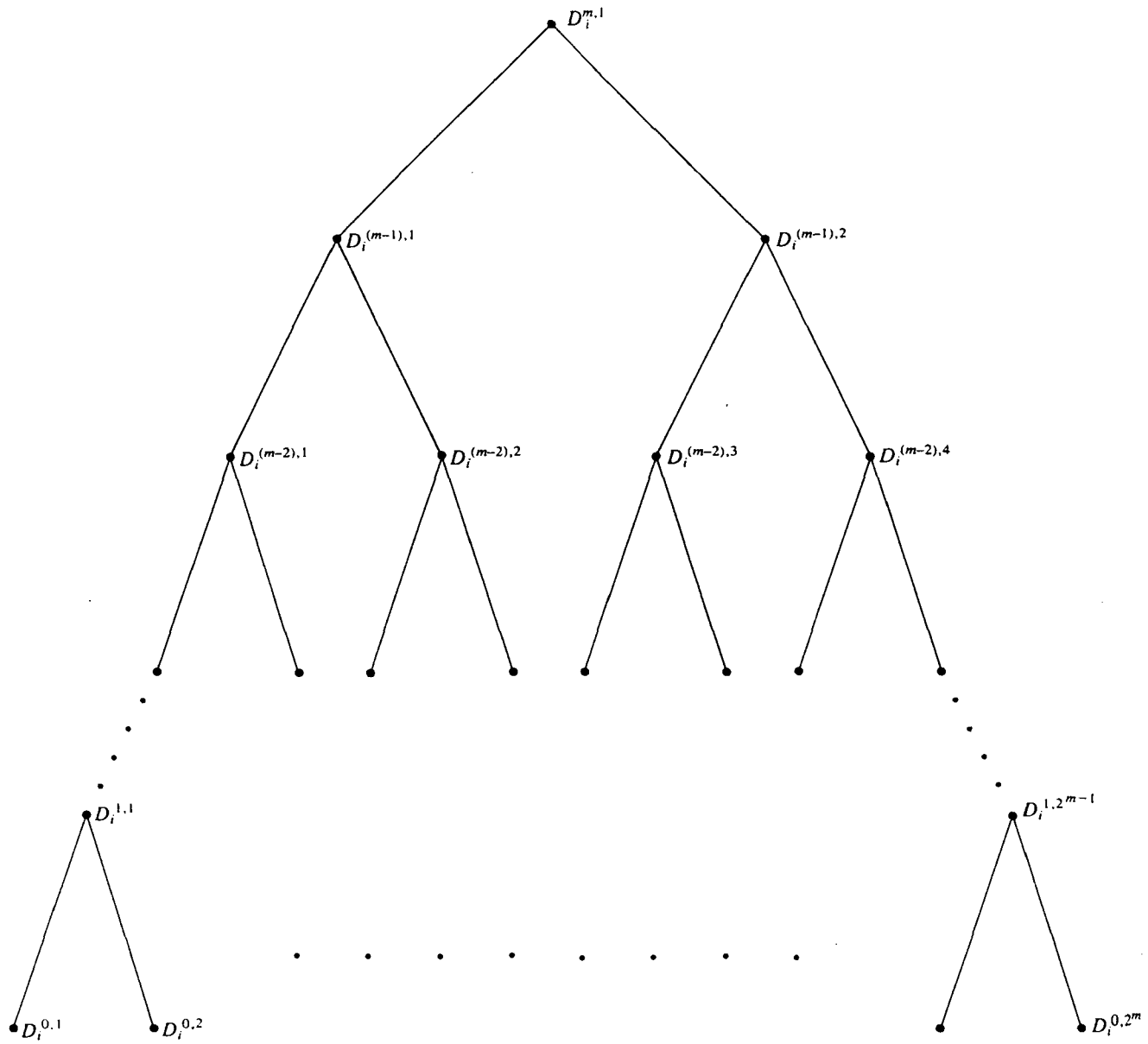

FIG. 1. Nomenclature for the domain tree containing subdomains of D_i.

We are given the predicate $P_{12}(a_s, b_t)$ as a relation matrix.

$$P_{12}: \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{array} \begin{array}{c} \begin{array}{cccc} b_1 & b_2 & b_3 & b_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

We have $A_{12}^{00} = P_{12}$.

Then we compute A_{12}^{01} by ORing together pairs of columns of A_{12}^{00} using [2]. The first column of A_{12}^{01} is obtained by bitwise ORing the first and second columns of A_{12}^{00}; the second column of A_{12}^{01} is obtained by ORing the third and fourth columns of A_{12}^{00}.

$$A_{12}^{01}: \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

and then we compute A_{12}^{02} from A_{12}^{01}

$$A_{12}^{02}: \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Using [3], A_{12}^{10} results from pairwise ANDing the rows of A_{12}^{00}.

$$A_{12}^{10}: \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Similarly, A_{12}^{11} results from pairwise ANDing the rows of A_{12}^{01}.

$$A_{12}^{11}: \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Parenthetically, for those readers somewhat bemused by the thicket of formal notation, let us remind you what this means.

For example, the entry in the first row of the second column of A_{12}^{11} is a 0, which indicates it is not true that for all elements of $D_1^{11} = \{a_1, a_2\}$ there is an element of D_2^{12} such that P_{ij} is satisfied.

A_{12}^{12} is computed by ANDing together the rows of A_{12}^{02}.

$$A_{12}^{12} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Finally, we obtain A_{12}^{20}, A_{12}^{21}, and A_{12}^{22} by ANDing the rows of A_{12}^{10}, A_{12}^{11}, and A_{12}^{12}, respectively.

A_{12}^{20}: [0 0 0 0]

A_{12}^{21}: [0 0]

A_{12}^{22}: [1]

We also define a set of predicates S_{ij}^{kl}, where $S_{ij}^{kl}(D_i^{ks}, D_i^{lt})$ is true iff for some member of abstract label D_i^{ks} there is a member of abstract label D_j^{lt} compatible with it.

$$S_{ij}^{kl}(D_i^{ks}, D_j^{lt}) \text{ iff } (\exists\, x \in D_i^{ks})(\exists\, y \in D_j^{lt}) P_{ij}(x,y)$$

We can also compute S_{ij}^{kl} inductively on k and l from the base predicate P_{ij} as follows:

$$S_{ij}^{00}(D_i^{0s}, D_j^{0r}) = P_{ij}(a_s, b_t)$$

where

[4] $D_i^{0s} = \{a_s\}$ and $D_j^{0t} = \{b_t\}$

[5] $S_{ij}^{0l}(D_i^{0s}, D_j^{lt}) = S_{ij}^{0(l-1)}(D_i^{0s}, D_j^{(l-1)(2l-1)}) \vee S_{ij}^{0(l-1)}(D_i^{0s}, D_j^{(l-1)2l})$ for $l = 1, \ldots, m$

and

[6] $S_{ij}^{kl}(D_i^{ks}, D_j^{lt}) = S_{ij}^{(k-1)l}(D_i^{(k-1)(2s-1)}, D_j^{lt}) \vee S_{ij}^{(k-1)l}(D_i^{(k-1)2s}, D_j^{lt})$

In other words the hierarchy of S_{ij} predicates is computed by collapsing P_{ij}: ORing together pairs of rows or pairs of columns. For our example:

$$S_{12}^{00}: \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$S_{12}^{01}: \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

$$S_{12}^{02}: \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$S_{12}^{10}: \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$S_{12}^{11}: \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$S_{12}^{12}: \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

S_{12}^{20}: [1 1 1 0]

S_{12}^{21}: [1 1]

S_{12}^{22}: [1]

We distinguish the arc (i,j) from the arc (j,i). Observe that

$$P_{ji} = (P_{ij})^{\mathrm{T}}$$

and so

$$A_{ji}^{00} = (A_{ij}^{00})^{\mathrm{T}}$$

and

$$S_{ij}^{00} = (S_{ij}^{00})^{\mathrm{T}}$$

It is always the case that $S_{ji}^{lk} = (S_{ij}^{kl})^{\mathrm{T}}$; this fact can be exploited computationally. However, A_{ji}^{lk} is, in general, *not* equal to $(A_{ij}^{kl})^{\mathrm{T}}$. In our example:

$$A_{21}^{00} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A_{21}^{01} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}$$

$$A_{21}^{11} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

whereas

$$(A_{12}^{11})^{\mathrm{T}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

5. Hierarchical arc consistency

We are now in a position to define a generalized hierarchical arc consistency algorithm to be known as HAC. HAC uses the general relaxation structure of AC-3 but uses REVISE-HAC as an instantiation of REVISE-A. In particular, step 3 of REVISE-A is

3 $\Delta \leftarrow \{x \mid (x \in \Delta_i) \wedge [(\exists\, y)(y \in \Delta_j) \wedge P_{ij}(x,y)\}$

where

$$\Delta_i = \bigcup_q \Delta_i^q \quad \text{and} \quad \Delta_j = \bigcup_r \Delta_j^r$$

as described earlier.

Step 3 can be implemented using the hierarchical predicates A and S defined above. Δ_i is represented by a list of its abstract labels Δ_i^q; Δ_j is similarly represented.

```
1  procedure REVISE-HAC ((i,j)):
2  begin
3  DELETE ← false
4  Q1 ← Δi
5  Δi ← ∅
6  while Q1 not empty do
7    begin
8    select and delete an element Di^ks from Q1
9    Q2 ← Δj
10   FOUND ← false
11   while Q2 not empty and not FOUND do
12     begin
13     select and delete an element Dj^lt from Q2
14     if Aij (Di^ks, Dj^lt) then
15       begin
16       Δi ← Δi ∪ {Di^ks}
17       FOUND ← true
18       end
19     end
```

```
20    if not FOUND then
21      begin
22      DELETE ← true
23      if k > 0 then
24        begin
25        Q2 ← Δj
26        while Q2 not empty and not FOUND do
27          begin
28          select and delete an element Djlt from Q2
29          if Sijkl(Diks, Djlt) then
30            begin
31            Q1 ← Q1 ∪ {Di(k-1)(2s-1), Di(k-1)2s}
32            FOUND ← true
33            end
34          end
35        end
36      end
37    end
38  return DELETE
39  end
```

REVISE-HAC: domain restriction for hierarchical arc consistency

```
    procedure NC (i):
    begin
    Δi ← Di ∩ {x | Pi(x)}
    end

1   begin
2   for i ← 1 until n do NC (i)
3   Q ← {(i,j) | (i,j) ∈ arcs (G), i ≠ j}
4   while Q not empty do
5     begin
6     select and delete any arc (k,m) from Q
7     if REVISE-HAC ((k,m))
8       then Q ← Q ∪ {i,k) | (i,k) ∈ arcs (G),i ≠ k,i ≠ m}
9     end
10  end
```

HAC: the hierarchical arc consistency algorithm

REVISE-HAC implements the generalized arc consistency algorithm introduced in Sect. 3. In particular, after the application of REVISE-HAC to arc (i,j), that arc will be strongly hierarchically arc consistent in the sense defined earlier. The loop defined in lines 6–37 tests each abstract label in Δ_i to see if it is hierarchically consistent. It does that by testing the label D_i^{ks} from Δ_i against the abstract labels in Δ_j. The loop in lines 11–19 looks for a label D_j^{lt} in Δ_j such that A_{ij} is true. If every label below D_i^{ks} is compatible with some label below D_j^{lt} then A_{ij} is true. In that case, D_i^{ks} survives unchanged in Δ_i. If not and D_i^{ks} is not a leaf ($k > 0$) then lines 20–36 look for a label in Δ_j such that some label below D_i^{ks} is compatible with some label below D_j^{lt} in which case S_{ij} is true. If such a label is found then the label D_i^{ks} is replaced by its two successors in Q_1. They must be tested similarly on A_{ij} and S_{ij} before this invocation of REVISE-HAC returns.

It should be shown that REVISE-HAC is correct and always terminates. Δ_i starts as the empty set and adds members only at line 16 when A_{ij} (D_i^{kl}, D_j^{lt}) is true and so when REVISE-HAC returns, all members of Δ_i are strongly arc consistent with some abstract label in Δ_j; hence the algorithm is correct. Since the domain trees are noncyclic and the queues Q_1 and Q_2 decrease monotonically in size (except at line 28 where a finite total number of elements can be added to Q_2), the procedure must terminate. The serial relaxation algorithm that has the form of the modified AC-3 but uses REVISE-HAC is known as HAC.

6. Complexity results

The algorithm AC-3 requires time linear in the number of constraints. As we remarked earlier it is at best $\Omega(a^2e)$ and at worst $O(a^3e)$. The unit of time used is the evaluation of a binary predicate on a pair of domain elements. We should not expect HAC to improve on the worst case performance of AC-3. Indeed, since it relies on a hierarchical organization of the domain, intuition suggests that one could perversely structure the domains in the worst possible way to ensure worst case behaviour worse than AC-3.

We first consider the time required to compute the hierarchical predicates A_{ij}^{kl} and S_{ij}^{kl} defined by the recursive eqs. [1]–[6]. Computation of these predicates is a preprocessing step required by HAC, if they are not already provided for an application. For each arc (i,j) we must compute A_{ij}^{kl} and S_{ij}^{kl} where $k,l = 0,1,2,\ldots, m$. Consider that A_{ij}^{kl} is represented by a relation matrix, with $2^{m-k} \times 2^{m-l}$ entries. The total number of entries for A_{ij} is then

$$\sum_{k=0}^{m}\sum_{l=0}^{m} 2^{m-k}2^{m-l} = \sum_{p=0}^{m}\sum_{q=0}^{m} 2^p 2^q = \left(\sum_{p=0}^{m} 2^p\right)^2 = (2^{m+1}-1)^2$$

Each entry requires two predicate evaluations to compute except the 2^{2m} entries in A_{00} which require one. Similarly for S_{ij}^{kl}; so for arc (i,j) the number of evaluations is

$$2(2(2^{m+1}-1)^2 - 2^{2m}) = 2^{2m+4} - 2^{2m+1} - 2^{m+4} + 4 \sim 14 \times 2^{2m}$$

Since $a = 2^m$ this is $14a^2$. Let e be the number of edges in the constraint graph. There are $2e$ arcs and so the preprocessing computation requires $\theta(ea^2)$, but this can be done once and for all for an application domain before any particular CSP is tackled.

The space required to store the A_{ij} and S_{ij} predicates is $\sim 12ea^2$ bits, a constant factor larger than the ea^2 bits required for the P_{ij} predicates. The preprocessing time and the storage space required can be reduced by exploiting facts such as $S_{ji} = (S_{ij})^T$ and the rule that if an entry in A_{ij} is 1 the corresponding entry in S_{ij} must also be 1.

The best case for the time complexity of HAC clearly occurs when the network is already strongly hierarchically arc consistent. In that case it merely has to check that condition which requires exactly $2e$ predicate evaluations of A_{ij}^{mm} between the root node of each domain tree and the root node of its neighbouring domain tree. So HAC is $\Omega(e)$.

Our analysis of the HAC worst case behaviour parallels the analysis in Mackworth and Freuder (1985), so it will not be spelled out in detail. Let d_i be the edge degree of vertex i in the constraint graph and let n be the number of vertices (variables). The worst case would occur when there is no solution, but that fact is discovered in the slowest possible way. For each variable x_i, REVISE-HAC can minimally replace one of the abstract labels by its two successors (or deleting it if it is a leaf in the domain tree). When that occurs, $(d_i - 1)$ arcs are, at worst, added to HAC's arc queue Q. The domain size is the number of leaves in the domain tree, $a = 2^m$; therefore that replacement or deletion can occur $2a - 1$ times since there are that many abstract labels in the domain tree.

The number of arcs that are, in total, then removed from Q is the number of arcs originally on Q plus the number added to Q

as a result of REVISE-HAC modifying a domain:

$$2e + \sum_{i=1}^{n} (2a - 1)(d_i - 1) = 2e + (2a - 1)(2e - n)$$

For each arc (i,j) the number of predicate evaluations is at worst the product of the current sizes of the two abstract label sets. Notice that there are a leaf vertices in the domain tree and $a - 1$ interior vertices, for a total of $(2a - 1)$. Since no abstract label can be active at the same time as any of its descendants or ancestors, the number of abstract labels active cannot exceed a. Notice that of the a labels, $(a - 1)$ each require up to a predicate evaluations of A_{ij} and the single label that is deleted or replaced by its two successors requires up to $4a$: A_{ij} and S_{ij} on the label itself and A_{ij} on its two successors. Since we assume $a \gg 1$ we count that as a^2 evaluations. Accordingly, the number of predicate evaluations is, at most,

$$a^2[2e + (2a - 1)(2e - n)]$$

We may, without loss of generality, assume that $e \geq n - 1$ (Mackworth and Freuder 1985), and so the time complexity of HAC is $O(a^3e)$. Since the complexity of HAC is asymptotically $4a^3e$ compared with AC-3's $2a^3e$, the intuition that the worst case for HAC is worse than the worst case for AC-3 is confirmed and quantified: it may be twice as slow. Another way to approach this is to realize that we have essentially doubled the domain size from a to $(2a - 1)$ by adding the interior nodes so the number of possible deletions from the domain has doubled. However, since only a labels can be active at once the number of predicate evaluations is still only a^2 (not $(2a)^2 = 4a^2$) to test consistency at any iteration and so it is only twice as slow, not eight times!

Although, pessimistically, the worst case analysis of HAC suggests it can be twice as slow as AC-3, remember that our optimistic best case analyses show it to be faster by a factor of a^2. Rather than try to characterize average case performance, we shall analyze worst case performance of HAC on problems for which it was designed.

A more reasonable analysis of HAC would consider those applications in which the domains *are* appropriately structured. A way to characterize this is to require that there is only one abstract label active in each node's domain at anytime, that is, $|\Delta_i| \leq 1$. Intuitively, one can think then of the variable's domain being progressively refined and reduced by the evidence of its related neighbouring variables. The specialization hierarchy is then being used as a true discrimination tree. If this is the case then a similar worst case analysis proceeds as follows. If REVISE-HAC returns true on (i,j) it has minimally replaced an abstract label by its successor. That can occur $m = \log_2 a$ times for that variable domain. Each time it occurs it adds $(d_i - 1)$ arcs to Q. The total number of arcs removed from Q is

$$2e + \sum_{i=1}^{n} (d_i - 1) \log a = 2e + (2e - n) \log a$$

For each arc removed an application of REVISE-HAC is needed, so the number of calls to REVISE-HAC is at most $2e + (2e - n) \log a$. We distinguish now between the successful calls to REVISE-HAC, on which a domain revision occurs and REVISE-HAC returns *true*, and the unsuccessful calls on which no change occurs and REVISE-HAC returns *false*. On the successful calls, REVISE-HAC tests A_{ij} on the single label in Δ_i and the label in Δ_j. That fails. S_{ij} succeeds. The label in Δ_i is replaced by its two successors: on one of them A_{ij} succeeds and on the other A_{ij} fails and S_{ij} fails. In all, five predicate evaluations are required. On the unsuccessful calls, only one evaluation of A_{ij} is required. The number of successful calls to REVISE-HAC is simply the number of possible deletions in a domain ($\log a$) times the number of domains (n). The number of unsuccessful calls to REVISE-HAC is the number of calls,

TABLE 1. P_{ij} for Geo-system surrounds Shore

P_{ij}	Lakeshore	Coastline
Island	1	0
Mainland	1	0
Lake	0	1
Ocean	0	1

TABLE 2. A_{ij} and S_{ij} for Geo-system surrounded by Shore

A_{ij}^{00}	Lakeshore	Coastline
Island	1	0
Mainland	1	0
Lake	0	1
Ocean	0	1

A_{ij}^{10}	Lakeshore	Coastline
Landmass	1	0
Waterbody	0	1

S_{ij}^{10}	Lakeshore	Coastline
Landmass	1	0
Waterbody	0	1

A_{ij}^{11}	Shore
Landmass	1
Waterbody	1

S_{ij}^{11}	Shore
Landmass	1
Waterbody	1

A_{ij}^{20}	Lakeshore	Coastline
Geo-system	0	0

S_{ij}^{20}	Lakeshore	Coastline
Geo-system	1	1

A_{ij}^{21}	Shore
Geo-system	0

S_{ij}^{21}	Shore
Geo-system	1

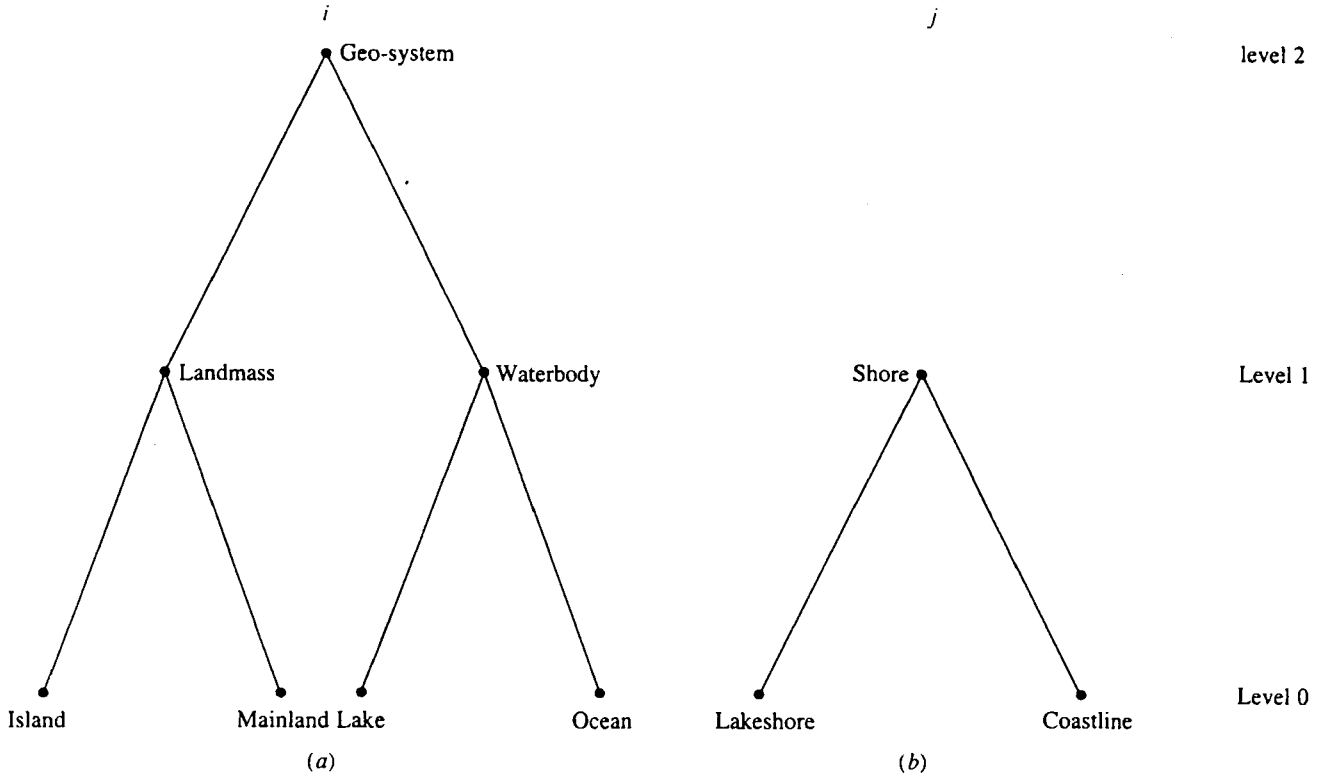

FIG. 2. (*a*) Geo-system specialization hierarchy and (*b*) Shore specialization hierarchy.

$2e + (2e - n) \log a$, minus the number of successful calls, $n \log a$. So the number of predicate evaluations is

$$5(n \log a) + 1[2e + (2e - n) \log a - n \log a] = 2e + (2e + 3n) \log a$$

And so, the worst case complexity of HAC under the specified condition is $O(\log a\ (e + \frac{3}{2}n))$, a remarkable improvement over AC-3's $O(a^3e)$.

7. Applications

HAC has been implemented and used in Mapsee3 (Mulder 1985), a schema-based system for interpreting hand-drawn sketch maps. A brief note here on how it is used should be useful. Schema instances represent scene objects and correspond to what we have called variables here. As a schema instance acquires more evidence as to its nature by acquiring a new component, for example, it can specialize its own interpretation; this action corresponds to moving down the domain tree. Moreover, instances that it is already related to may then be further specialized and so on. A geographical system (Geo-system) has the specialization hierarchy shown in Fig. 2*a* while a shoreline (Shore) has the specialization hierarchy of Fig. 2*b*.

Suppose a Geo-system completely surrounds a closed Shore in the map then we have the relation P_{ij} shown in Table 1.

In other words either the Geo-system is an Island or a Mainland and the Shore a Lakeshore or the Geo-system is a Lake or an Ocean and the Shore a Coastline. From this relation we can compute the hierarchical predicates A_{ij}^{kl} and S_{ij}^{kl} using [1]–[6]. They are shown in Table 2 (note that $S_{ij}^{00} = A_{ij}^{00}$).

Experimentally, for the sketch map application, Mulder (1985) found that HAC is more efficient than AC-3. On each of 10 maps HAC required fewer predicate evaluations than AC-3 in Mapsee3 by factors ranging up to about two, although here the domains are still very small: a varies from 2 to 8. Mulder also reported experimental evidence that the number of iterations does depend linearly on the number of constraints.

HAC is most useful when the domains can be naturally described hierarchically; that is, when the interior nodes of the domain tree are natural kinds. There must be nontrivial relationships between the total set of elements represented at an internal node of one domain tree and the total set of elements at an internal node of the neighbouring domain tree. Moreover, we expect the advantages of HAC to be more fully realized for very large domains.

The original edge labelling paradigm in which arc consistency was invented (Waltz 1972) is an example of such an application. Waltz essentially used the sets of possible corners as the variable domains with the edge type being the predicates. Interchanging the role of the corners and the edges so the variable domains are the edge types and the set of corners are the predicates, one can structure the very large number of edge types (1532) hierarchically (Mackworth 1977*c*) and use HAC.

To sketch how this would be done, consider the simple Huffman–Clowes world with only four edge types: convex, concave, and the two occluding edges. A natural grouping of convex and concave into the "connect" abstract label and the two occluding edges into the "occlude" abstract label yields a three-level edge domain hierarchy with the single abstract label "edge" at the root of the tree. The familiar catalogues of allowable corners for the junction shape categories correspond to the base predicates. Using the algorithms given earlier, the predicates families S and A can easily be constructed.

For example, a catalogue of allowable corner labellings at the connect/occlude abstract label level represents the S predicate between those levels of the domain trees. Since corners may involve two or three edges the version of HAC presented here for binary relations must be generalized to allow for n-ary relations. This can be done in the same way as AC-3 was generalized to n-ary relations in Mackworth (1977*b*).

8. Conclusions

A hierarchical arc consistency algorithm for constraint satisfaction problems, HAC, has been described that exploits the internal structuring of domain values into a hierarchy of subdomains. Complexity results show that the algorithm has demonstrably improved best and worst case performance if the domains obey certain constraints. In that case HAC is at best $\Omega(e)$ and at worst $O((e + \frac{2}{3}n) \log a)$ compared with $\Omega(a^2e)$ and $O(a^3e)$ for the previously best-known algorithm. Experimental results from the use of the algorithm in a computational vision system, Mapsee3, are consistent with our analysis.

9. Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada operating grants A9281 and A5502 and the Canadian Institute for Advanced Research. Alan Mackworth is a Fellow of the Canadian Institute for Advanced Research. We are grateful to May Vink for a heroic job of formatting the paper and to the referees for their careful critiques and useful suggestions.

FREUDER, E. C. 1978. Synthesizing constraint expressions. Communications of the ACM, **21**, pp. 958–966.

——— 1982. A sufficient condition for back track-free search. Journal of the ACM, **19**, pp. 24–32.

HARALICK, R. M., and ELLIOTT, G. L. 1980. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, **14**, pp. 263–313.

HAVENS, W. S., and MACKWORTH, A. K. 1983. Representing knowledge of the visual world. IEEE Computer, **16**(10), pp. 90–96.

MACKWORTH, A. K. 1977*a*. Consistency in networks of relations. Artificial Intelligence, **8**, pp. 99–118.

——— 1977*b*. On reading sketch maps. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, MIT, Cambridge, MA, pp. 598–606.

——— 1977*c*. How to see a simple world. Machine Intelligence. Vol. 8. *Edited by* E. W. Elcock and D. Michie. Wiley, New York, NY, pp. 510–537.

MACKWORTH, A. K., and FREUDER, E. C. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artificial Intelligence, **25**, pp. 65–74.

MAIER, D. 1983. The theory of relational databases. Computer Science Press, Rockville, MD.

MONTANARI, U. 1974. Networks of constraints: fundamental properties and applications in picture processing. Information Science, **7**, pp. 95–132.

MULDER, J. A. 1985. Using discrimination graphs to represent visual knowledge. Ph.D. thesis (TR 85-14), Department of Computer Science, University of British Columbia, Vancouver, B.C.

WALTZ, D. E. 1972. Generating semantic descriptions of scenes with shadows. Technical Report MAC AI-TR-271, MIT, Cambridge, MA.