Consistency in Networks of Relations

Alan K. Mackworth

Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada

Recommended by Saul Amarel

ABSTRACT

Artificial intelligence tasks which can be formulated as constraint satisfaction problems, with which this paper is for the most part concerned, are usually solved by backtracking. By examining the thrashing behavior that nearly always accompanies backtracking, identifying three of its causes and proposing remedies for them we are led to a class of algorithms which can profitably be used to eliminate local (node, arc and path) inconsistencies before any attempt is made to construct a complete solution. A more general paradigm for attacking these tasks is the alternation of constraint manipulation and case analysis producing an OR problem graph which may be searched in any of the usual ways.

Many authors, particularly Montanari and Waltz, have contributed to the development of these ideas; a secondary aim of this paper is to trace that history. The primary aim is to provide an accessible, unified framework, within which to present the algorithms including a new path consistency algorithm, to discuss their relationships and the many applications, both realized and potential, of network consistency algorithms.

1. Introduction

A concern for the efficiency of our programs is not a major component of the current artificial intelligence zeitgeist, and yet, as the focus shifts from small, toy problems to large ones, that concern should become more central. Even if, as some claim by way of excuse, the technology is advanced at an exponential rate, if our programs consume a quantity of resources that is exponential in the size of the task, $O(k^n)$, then each doubling of available resources only means an additional ($\ln 2/\ln k$) words, regions or clauses can be handled. This paper is concerned with the effectiveness of algorithms designed to solve a certain class of problems.

2. The Task

Many tasks can be seen as constraint satisfaction problems. In such a case the task specification can be formulated to consist of a set of variables, each of which must be instantiated in a particular domain and a set of predicates that the values of the variables must simultaneously satisfy. Restricting the discussion for the moment to unary and binary predicates the task consists, then, of providing a constructive proof for the wff:

$$(\exists x_1)(\exists x_2)\ldots(\exists x_n)P_1(x_1) \wedge P_2(x_2) \wedge \ldots \wedge P_n(x_n) \wedge P_{12}(x_1, x_2) \wedge P_{13}(x_1, x_3) \wedge \ldots \wedge P_{n-1,n}(x_{n-1}, x_n),$$

where P_{ij} is only included in the wff if $i < j$ for we require $P_{ji}(v_j, v_i) = P_{ij}(v_i, v_j)$. Imposing a further restriction that the variable domains each consist of a finite number of discrete values then there are several candidate solution schemes. Among these are generate-and-test, formal theorem-proving methods and backtracking.

3. Backtracking and Three of its Maladies

Backtracking consists, in general, of the sequential instantiation of the variables from ordered representations of their domains. As soon as all of the variables of any predicate are instantiated its truth value is tested. If it is true the process of instantiation and testing continues but if it is false the process fails back to the last variable instantiated that has untried values in its domain and reinstantiates it to its next value. The intrinsic merit of backtracking is that substantial subspaces of the generate-and-test search space, the Cartesian product of all the variable domains, are eliminated from further consideration by a single failure.

On the other hand, backtracking can still be grotesquely inefficient. See Sussman and McDermott [19] and Gaschnig [10] for particular samples of pathological behavior. Bobrow and Raphael [2] have labeled this class of behavior "thrashing". In particular, the time taken to find a solution tends to be exponential in the number of variables both in the worst-case and on the average. It is important to identify the causes of this poor behavior and to suggest remedies.

(A) The most obvious source of inefficiency and the easiest to prevent concerns the unary predicates. If the domain for variable v_i, D_i, includes a value that does not satisfy $P_i(x)$ then it will be the cause of repeated instantiation and failure which could be eliminated by simply discarding once and for all those domain elements that do not satisfy the corresponding unary predicate.

(B) A second source of inefficiency occurs in the following situation. Suppose the variables are instantiated in the order $v_1, v_2, \ldots, v_n$ and for $v_i = a$, $P_{ij}(a, v_j)$ (where $j > i$) does not hold for any value of v_j. Backtracking will try all values of v_j, fail and try all values of v_{j-1} (and for each of these try all values of v_j) and so on until it tries all combinations of values for $v_{i+1}, v_{i+2}, \ldots, v_j$ before finally discovering that a is not a possible value for v_i. What's worse this identical failure process may be repeated for all other sets of values for $v_1, v_2, \ldots v_{i-1}$ with $v_i = a$.

(C) A third phenomenon that causes gross inefficiency and replication of effort occurs when $v_i = a$, $v_j = b$, and $P_i(a)$, $P_j(b)$ and $P_{ij}(a, b)$ do hold but there is no value x for a third variable v_k such that $P_{ik}(a, x)$, $P_k(x)$ and $P_{kj}(x, b)$ are simultaneously satisfied as they must be in any solution. As in the previous case, this is not only expensive to discover but may also be rediscovered many times by a backtracking solution process.

It is the purpose of this paper to provide a unified treatment of these phenomena and algorithms designed to prevent their occurrence thereby leading to solution strategies that do not require exponential time for particular task domains.

It is convenient to view the task specification as a network which consists of a labeled, directed graph in which the variables are represented by the nodes each with an associated set representing the variable's domain, the unary predicates are represented by loops on the nodes, and the binary predicates by labeled, directed arcs. For each arc from node i to node j corresponding to $P_{ij}(i < j)$ there is an arc from node j to node i corresponding to $P_{ji}(v_j, v_i) = P_{ij}(v_i, v_j)$. Suitable terms to name each of the state of affairs that lead to the three phenomena described above are, respectively, node inconsistency, arc inconsistency, and path inconsistency.

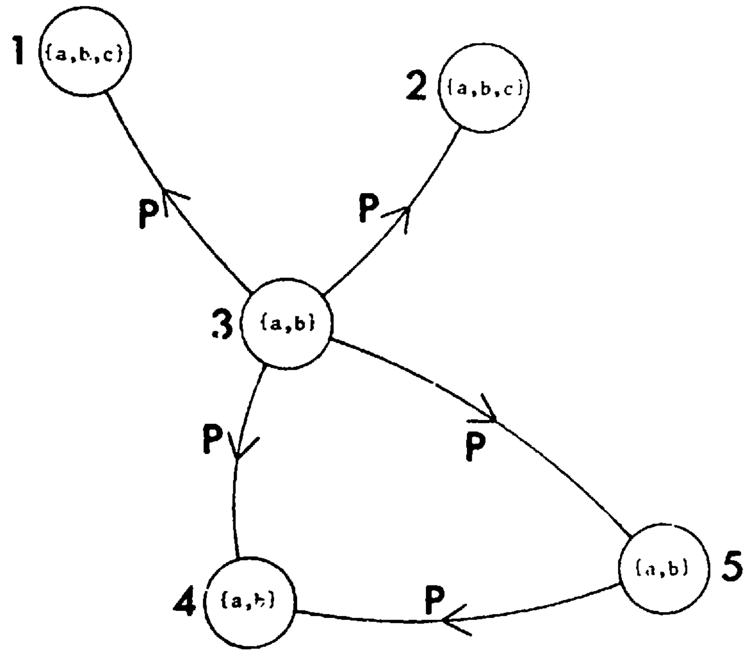


FIG. 1. A network exhibiting inconsistency.

As an example of arc and path inconsistencies consider the network of Fig. 1. The domains associated with the nodes are $D_1 = D_2 = \{a, b, c\}$ and $D_3 = D_4 = D_5 = \{a, b\}$. The binary predicate P denotes strict lexicographic ordering of its arguments so that $P(a, b) = P(a, c) = \ldots = T$ and $P(a, a) = P(b, a) = \ldots = F$. The arcs for $P^T(x, y) = P(y, x)$ are omitted. A backtrack search demonstrating that no solution exists is shown in Fig. 2. The network nodes are treated in the order 1, 2, 3, 4, 5. Each node in the search tree is labeled with the partial solution developed to that node. As usual, as soon as a partial solution fails to satisfy one of the network relations backtracking occurs.

Arc inconsistency appears at, for example, arc 2–3. The value $a \in D_2$ has no corresponding $x \in D_3$ such that $P(x, a)$. In the search tree this is reflected as one of the reasons for the failure of the subtree rooted at *aa*, (*aa*(*aaa*)(*aab*)), and rediscovered in the failure of the subtree (*ba*(*baa*)(*bab*)) and again in the subtree (*ca*(*caa*)(*cab*)).

```
a
  aa
      aaa
      aab
  ab
      aba
      abb
  ac
      aca
      acb
b
  ba
      baa
      bab
  bb
      bba
          bbaa
          bbab
                bbaba
                bbabb
      bbb
  bc
      bca
          bcaa
          bcab
                bcaba
                bcabb
      bcb
c
  ca
      caa
      cab
  cb
      cba
          cbaa
          cbab
                cbaba
                cbabb
      cbb
  cc
      cca
          ccaa
          ccab
                ccaba
                ccabb
      ccb
          ccba
          ccbb
```

FIG. 2. A backtracking search of the solution space of the network in Fig. 1.

Path inconsistency appears on the path 3–5–4. The values $a \in D_3$ and $b \in D_4$ satisfy the vacuous unary predicates and the non-vacuous binary predicate $P(a, b)$; however, there is no value $x \in D_5$ such that $P(a, x) \wedge P(x, b)$. This is discovered in the search tree in the failure of the subtree (*bbab*(*bbaba*)(*bbabb*)) and rediscovered three more times in the failure of the subtrees rooted at nodes *bcab*, *cbab*, and *ccab*.

This example was chosen to be as small as possible and yet still demonstrate these effects. Clearly with larger domains and larger networks these problems multiply. In particular, in this example there are no nodes intervening between the nodes causing the failure. If there are such intervening nodes that are irrelevant to the failure then the failed subtrees can be very much larger besides reoccurring often. It should also be clear that although here the ordering of the nodes was somewhat malicious in intent (although it could have been worse: consider 1, 2, 4, 5, 3) these inefficiencies of backtracking cannot be removed by such minor palliatives as reordering the nodes.

4. Consistency: A State of Affairs that Forestalls Thrashing

The state of affairs that ensures that those phenomena do not occur can be defined as follows:

(A) *Node consistency*

Node i is node consistent iff for any value $x \in D_i$, $P_i(x)$ holds.

(B) *Arc consistency*

Arc (i, j) is arc consistent iff for any value $x \in D_i$ such that $P_i(x)$, there is a value $y \in D_j$ such that $P_j(y)$ and $P_{ij}(x, y)$.

(C) *Path consistency*

A path of length m through the nodes $(i_0, i_1, \ldots, i_m)$ is path consistent iff for any values $x \in D_{i_0}$ and $y \in D_{i_m}$ such that $P_{i_0}(x)$ and $P_{i_m}(y)$ and $P_{i_0 i_m}(x, y)$, there is a sequence of values $z_1 \in D_{i_1}, \ldots, z_{m-1} \in D_{i_{m-1}}$ such that

(i) $P_{i_1}(z_1)$ and ... and $P_{i_{m-1}}(z_{m-1})$,

(ii) $P_{i_0 i_1}(x, z_1)$ and $P_{i_1 i_2}(z_1, z_2)$ and ... and $P_{i_{m-1} i_m}(z_{m-1}, y)$.

The definition and example of path inconsistency given in Section 3 was only for path length $m = 2$. This definition of path consistency does not require that the nodes $(i_0, i_1, \ldots, i_m)$ all be distinct. That is, path consistency applies to both simple and non-simple paths. Moreover, a non-simple path may be consistent even though different occurrences of the same node on the path correspond to different occurrences of the associated variable.

A network is said to be node, arc or path consistent iff every node, arc or path of its graph is consistent.

5. How to Achieve Node Consistency

Since node consistency is concerned only with the unary predicates, in achieving it there is no interaction between the nodes; thus, it is achieved by a simple one-pass algorithm NC-1 that applies the node consistency procedure NC to each node i.

```
**procedure** NC($i$):
$D_i \leftarrow D_i \cap \{x \mid P_i(x)\}$
**begin**
    **for** $i \leftarrow 1$ **until** $n$ **do** NC($i$)
**end**
```

NC-1: the node consistency algorithm

6. How to Achieve Arc Consistency

The algorithms in this section are all based on the following observation (first made by Fikes [6]): given discrete domains, D_i and D_j, for two variables v_i and v_j which are node consistent, if $x \in D_i$ and there is no $y \in D_j$ such that $P_{ij}(x, y)$ then x can be deleted from D_i. When that has been done for each $x \in D_i$ then arc (i, j) (but not necessarily (j, i)) is consistent. As this is the basic action of the arc consistency algorithms we embody it in a Boolean procedure:

```
procedure REVISE($(i, j)$):
begin
  DELETE ← false
  for each $x \in D_i$ do
    if there is no $y \in D_j$ such that $P_{ij}(x, y)$ then
      begin
        delete $x$ from $D_i$;
        DELETE ← true
      end;
  return DELETE
end
```

Note that immediately after applying REVISE to arc (i, j) it must be consistent; however, it may not remain consistent because values in D_j may subsequently be removed by applications of REVISE to some arc (j, k). A single pass through all the arcs applying REVISE to each is not sufficient. The simplest algorithm to achieve arc consistency, AC-1, iterates such a pass until there is no change on an entire pass at which point the network must be arc consistent.

```
begin
  for $i \leftarrow 1$ until $n$ do NC($i$);
  $Q \leftarrow \{(i, j) \mid (i, j) \in \text{arcs}(G), i \neq j\}$
  repeat
    begin
      CHANGE ← false
      for each $(i, j) \in Q$ do CHANGE ← (REVISE ($(i, j)$) or CHANGE)
    end
  until ¬ CHANGE
end
```

AC-1: the first arc consistency algorithm

The obvious inefficiency in AC-1 is that a single, successful revision of an arc on a particular iteration causes all the arcs to be revised on the next iteration whereas in fact only a small fraction of them could possibly be affected.

In noting this fact Waltz [24] implemented an elegant algorithm that he described as follows: (to convert to our framework, for "junction" read "node", for "label" read "value" and for "branch" read "arc").

"[The result] is obtained by going through the junctions in numerical order and:

(1) Attaching to a junction all labels which do not conflict with junctions previously assigned, i.e., if it is known that a branch must be labeled from the set S, do not attach any junction labels which would require that the branch be labeled with an element not in S.

(2) Looking at the neighbors of this junction which have already been labeled; if any label does not have a corresponding assignment for the same branch, then eliminate it.

(3) Whenever any label is deleted from a junction, look at all its neighbors in turn, and see if any of their labels can be eliminated. If they can, continue this process iteratively until no more changes can be made. Then go on to the next junction (numerically)."

The idea behind this algorithm is that arc consistency can be achieved in one pass through the nodes by ensuring that following the introduction of node i all arcs (k, m) where $k, m \leqslant i$ and $k \neq m$ are made consistent. When node $i+1$ is introduced all arcs leading from it and all arcs leading to it (to and from nodes introduced earlier) may be inconsistent and so must be revised. If a REVISE((k, m)) is successful (i.e., modifies D_k) then the only additional arcs that need to be reconsidered are all those that lead to k, $\{(p, k)\}$, with the important exception of (m, k). (m, k) is excepted because it cannot have become inconsistent as a direct result of the deletions made in D_k by REVISE((k, m)): any deletions were made precisely because there was no corresponding value in D_m.

These notions are captured in AC-2 which follows that Waltz' filtering algorithm in spirit (see p. 106).

When node i is introduced on the ith iteration of lines 3–18, Q and Q' are initialized on lines 5 and 6 to contain all arcs directed away from and toward node i respectively. When Q is exhausted by the iteration of lines 10–14, Q is set to Q' and Q' emptied ready to hold all arcs directed at nodes one arc removed from node i that need to be revised. At the start of the sth ($s > 2$) iteration of lines 8–17, Q consists of all arcs directed at nodes $(s-2)$ arcs removed from i that are to be revised while Q' is ready to hold all the arcs directed at nodes $(s-1)$ arcs removed from i that need to be revised as a result of the revising of the arcs on Q. This process initially spreads out from node i but may return to it if there are any cycles in the graph of arc length greater than 2. The particular form of AC-2 derives, in part, from considering just such a situation in which the spreading wave of arc revision will cross itself.

Another approach to the arc consistency problem abandons the idea of making the network arc consistent on a single pass through the nodes. Instead simply make a queue of all the arcs in the network and apply REVISE to them sequentially. If

```
begin
  for $i \leftarrow 1$ until $n$ do
    begin
      NC($i$);
      $Q \leftarrow \{(i,j) \mid (i,j) \in \text{arcs}(G), j < i\}$
      $Q' \leftarrow \{(j,i) \mid (j,i) \in \text{arcs}(G), j < i\}$
      while $Q$ not empty do
        begin
          while $Q$ not empty do
            begin
              pop $(k,m)$ from $Q$
              if REVISE($(k,m)$) then
                $Q' \leftarrow Q' \cup \{(p,k) \mid (p,k) \in \text{arcs}(G), p \leqslant i, p \neq m\}$
            end
          $Q \leftarrow Q'$
          $Q' \leftarrow$ empty
        end
    end
end
```

AC-2: the second arc consistency algorithm

REVISE is successful on any arc (reduces a node domain) then one need only (re)apply REVISE to those arcs that could possibly have the result of applying REVISE changed from *false* to *true*. (Contrast this with AC-1 which would subsequently reapply REVISE to all the arcs.) Some of these arcs may already be waiting on the queue. If so, they should not be reentered on it. AC-3 embodies this approach.

```
begin
  for $i \leftarrow 1$ until $n$ do NC($i$);
  $Q \leftarrow \{(i,j) \mid (i,j) \in \text{arcs}(G), i \neq j\}$
  while $Q$ not empty do
    begin
      select and delete any arc $(k, m)$ from $Q$;
      if REVISE ($(k, m)$) then $Q \leftarrow Q \cup \{(i,k) \mid (i,k) \in \text{arcs}(G), i \neq k, i \neq m\}$
    end
end
```

AC-3: the third arc consistency algorithm

Although AC-2 appears more complex than AC-3 it is just a special case of the latter algorithm corresponding to the choice of a particular ordering of AC-3's priority queue, with the exception of one minor discrepancy. The discrepancy is that in AC-2 it is possible if the graph has cycles of arc length greater than 2 for an arc to be waiting on both Q and Q' simultaneously.

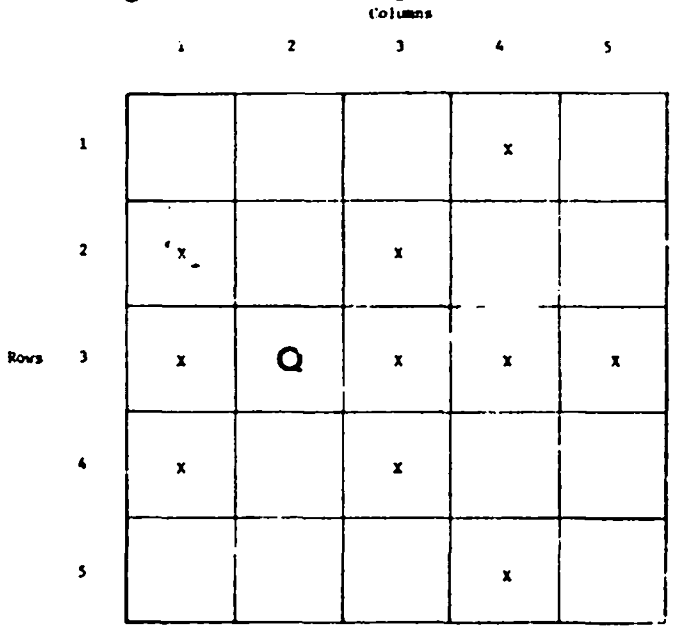
7. How to Achieve Path Consistency

Montanari [14] has provided an elegant, formal treatment of the concept of path consistency. The purpose of this section is to introduce some of Montanari's notation and theorems and his algorithm for achieving path consistency and, furthermore, to show how the same result can be achieved by doing considerably less computation by refining the algorithm in a manner somewhat analogous to the progression from AC-1 to AC-3.

7.1. Representing relations

The arc consistency algorithms operate on an explicit data structure representation of the unary predicates, (i.e., the sets of all values that satisfy them, D_i) deleting values that cannot be part of a complete solution because of the restrictions imposed on adjacent nodes by the binary predicates. However, it is a matter of indifference to those algorithms whether the binary predicates are represented by a data structure or a procedure. The path consistency algorithms can be seen as generalizations in that although the predicate $P_{13}(x, y)$ may allow a pair of values, say, $P_{13}(a, b)$ that pair may actually be forbidden because there is an indirect constraint on v_1 and v_3 imposed by the fact that there must be a value, c, for v_2 that satisfies $P_{12}(a, c)$, $P_2(c)$ and $P_{23}(c, b)$. If there is no such value then that fact may be recorded by deleting the pair (a, b) from the set of value pairs allowed initially by P_{13}, in a fashion directly analogous to the deletion of individual values from the variable domains in the arc consistency algorithms. In order to perform that deletion it is necessary to have a data representation for the set of pairs allowed by a binary predicate. If the variable domains are finite and discrete then a relation matrix with binary entries is such a representation. Predicate P_{ij} is represented by a relation matrix R_{ij} whose m_i rows correspond to the m_i values of v_i and whose m_j columns correspond to the m_j values of v_j.

A useful example of the concepts involved is the set of n-queens problems. Used by Floyd [7], Fikes [6] and Djikstra [4] to illustrate backtrack programming, REF-ARF and structured programming respectively, this example is also of historical and comparative interest. The task is to place n queens on an $n \times n$ chessboard so that no queen is on the same row, column or diagonal as any other. Since each queen must be in a different column the task can be put in the constraint satisfaction paradigm by creating n variables $(v_1, v_2, \ldots, v_n)$, one for each column. The value of each variable is the row number of the queen in that column. Consider the 5-queens problem of Fig. 3(a). The queen shown in column 2, $(v_2 = 3)$, forbids the values $v_1 = 2$, $v_1 = 3$ and $v_1 = 4$ hence column 3 of the initial value of R_{12} is as shown in Fig. 3(b). For uniformity the currently permitted values for each variable are not given by a set D_i as for the arc consistency algorithms but by a matrix R_{ii} whose off-diagonal entries are required to be zero.



(a)

$$R_{12} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

(b)

FIG. 3. Illustrating the 5-queens problem.

7.2. Operations on relations

Two operations on relations are needed: intersection and composition.

7.2.1. *Intersection of relations*

If two separate relations are both required to hold between v_i and v_j, R'_{ij} and R''_{ij}, then their intersection is written $R_{ij} = R'_{ij}$ & R''_{ij} where the entry in the rth row and sth column of R_{ij}: $R_{ij,rs} = R'_{ij,rs} \wedge R''_{ij,rs}$.

7.2.2. *Composition of relations*

Suppose relation R_{12} holds between v_1 and v_2 and R_{23} between v_2 and v_3 then the induced relation transmitted by v_2 is the composite relation $R_{13} = R_{12} \cdot R_{23}$. A pair (a, c) is allowed by R_{13} only if there is a pair (a, b) allowed by R_{12} and a pair (b, c) allowed by R_{23}. That is,

$$R_{13} = R_{12} \cdot R_{23}$$

iff

$$R_{13,rs} = \bigvee_{t=1}^{m_2} (R_{12,rt} \wedge R_{23,ts}).$$

In the matrix representation, composition is simply binary matrix multiplication. Composition of relation matrices takes precedence over intersection.

7.3. Direct and induced relations

If, in the example above, R^0_{13} was the original direct relation between v_1 and v_3 then it can be intersected with the induced relation $R_{12} \cdot R_{23}$ to give a new and possibly more restrictive constraint $R'_{13} = R^0_{13}$ and $R_{12} \cdot R_{23}$.

7.3.1. *Two examples of induced relations*

7.3.1.1. *5-queens.* In Fig. 4, R^0_{25} permits the pair of queens shown but there is no value of v_1 that satisfies both R_{21} and R_{15} so $R'_{25} = R^0_{25}$ and $R_{21} \cdot R_{15}$ forbids the pair of queens shown. In the matrix notation $R^0_{25,31} = 1$ but $(R_{21} \cdot R_{15})_{31} = 0$ hence $R'_{25,31} = 0$.

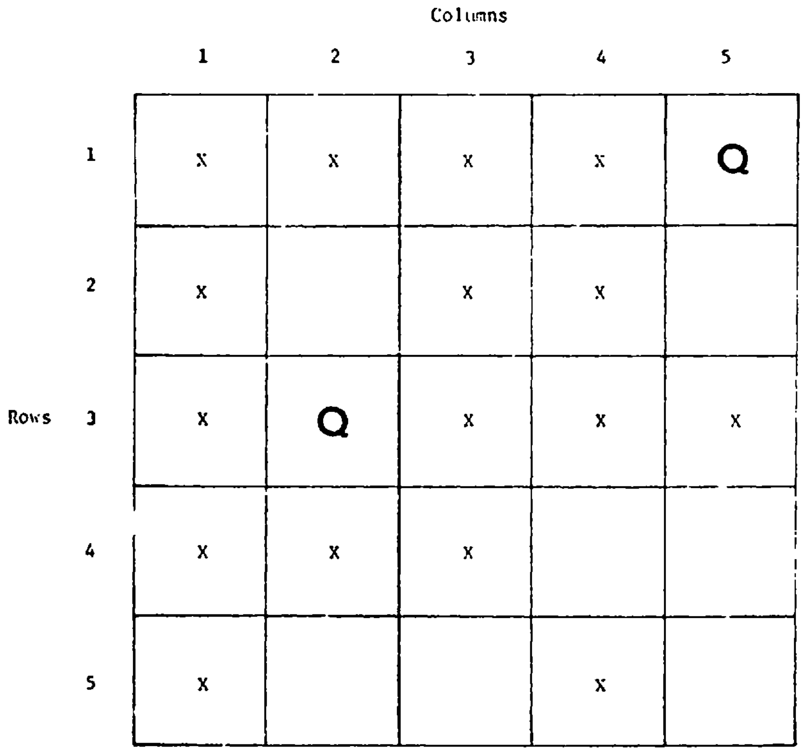


FIG. 4. Induced relations in the 5-queens problem.

7.3.1.2. *Arc consistency.* The basic arc consistency procedure of Section 6, REVISE $((i, j))$, can be written in the current notation as:

$$R'_{ii} = R^0_{ii} \;\&\; R_{ij} \cdot R_{jj} \cdot R_{ji} \tag{7.1}$$

To see that this is so, note that $R_{ji} = R^T_{ij}$ and $R_{jj} = R_{jj} \cdot R_{jj}$ so (7.1) can be written as

$$R'_{ii} = R^0_{ii} \;\&\; R_{ij} \cdot R_{jj} \cdot R^T_{ij}$$

$$R'_{ii} = R^0_{ii} \;\&\; (R_{ij} \cdot R_{jj}) \cdot (R_{ij} \cdot R_{jj})^T$$

Each row of R_{ij}, corresponding to each value of v_i, has a 1 for each value of v_j allowed. $R_{ij} \cdot R_{jj}$ is the same except that all columns corresponding to nonpermitted values of v_j will be zeroed. $(R_{ij} \cdot R_{jj}) \cdot (R_{ij} \cdot R_{jj})^T$ will have a 1 at position rr on the main diagonal iff $R_{ij} \cdot R_{jj}$ has at least one 1 in row r (that is, there is at least one value for v_j for the rth value of v_i). R_{ii}^0 is zero off the diagonal and 1 at position rr if the rth value of v_i was previously allowed; this position is zero in R'_{ii} iff there is no corresponding value of v_j. Thus the effect of (7.1) parallels exactly the side effect of REVISE((i, j)).

7.4. The minimal network

Having introduced the notion of induced relations it is natural to enquire if there is an algorithm that makes explicit all the induced relations implicit in a network. To specify the task properly we need two definitions:

(a) Two networks N_1 and N_2 each with n nodes are *equivalent* iff the set of n-tuples satisfying N_1 is identical to the set of n-tuples satisfying N_2

(b) A network M is *minimal* iff

$$(R_{ij,x_ix_j} = 1) \rightarrow (\exists v_1)(\exists v_2) \ldots (\exists v_n)(v_i = x_i)(v_j = x_j)(\forall k)(\forall p)(R_{kp,v_kv_p} = 1).$$

In English, in a minimal network the remainder of the network does not add any further constraint to the direct constraint R_{ij} between v_i and v_j. If any pair of values is permitted by its direct constraint then it is part of at least one solution. The task that Montanari calls the central problem is to compute for a given network N, a network M that is minimal and equivalent to N. The central problem is clearly solvable: generate the set of all solutions by backtracking and then for all i and j set $R_{ij,ab} = 1$ iff there is a solution $(x_1, x_2, \ldots, x_n)$ where $x_i = a$ and $x_j = b$. However, that is expensive.

In fact, the central problem is NP-complete, that is, putatively exponential. (Montanari [15] credits this observation to a private communication from R. M. Burstall.) It is easy to see that this must be so. If it is solvable in polynomial time then so is the problem of deciding if a planar, undirected graph with at most four edges incident at a node has a chromatic number of at most 3 which in turn is known to imply that P = NP (which conjecture is thought unlikely to be true) [9, 1]. The chromatic number of a graph is the minimum number of different colours needed to paint the nodes so that each node is a different colour from every adjacent node. To put the chromatic number problem into our framework, the relations R can all be 3×3 Boolean matrices. R_{ii} is the 3×3 identity matrix while R_{ij} $(i \neq j)$ is 0 on the main diagonal and 1 off it if there is an arc (i, j) otherwise the entries of R_{ij} $(i \neq j)$ are all 1. If the central problem for this network can be solved in polynomial time then one can simply inspect any R_{ii}: if there is a non-zero entry then the 3-colorability decision problem is answered affirmatively otherwise negatively. This sharp result, due to Garey, Johnson and Stockmeyer in [9], shows that even quite restricted network consistency problems can be inherently exponential; here, for example, we have domains of size 3 and only four non-vacuous relations out of each node.

7.5. Path consistency

Given that the central problem is not likely to admit of an efficient (polynomial time) solution, it seems judicious to attack an easier problem: the task of computing a path consistent network equivalent to a given network. To recall, a network is path consistent iff any pair allowed by any direct relation R_{ij} is also allowed by all paths from v_i to v_j. A pair is allowed by a path from v_i to v_j if at every intermediate vertex values can be found that satisfy the unary and binary predicates along the path. The following theorem due to Montanari [14] can be used to justify the first path consistency algorithm.

THEOREM. *If every path of length 2 of a network with complete graph is path consistent the network is path consistent.*

Proof. By straightforward induction on the length of the path.

Observe that in our notation a path of length 2 from node i through node k to node j is consistent iff $R_{ij} = R_{ij} \& R_{ik} \cdot R_{kk} \cdot R_{kj}$. The algorithm given by Montanari to compute a path consistent network equivalent to R is then as follows:

```
1  begin
2    $Y^n \leftarrow R$
3    repeat
4      begin
5        $Y^0 \leftarrow Y^n$
6          for $k \leftarrow 1$ until $n$ do
7            for $i \leftarrow 1$ until $n$ do
8              for $j \leftarrow 1$ until $n$ do
9                $Y^k_{ij} \leftarrow Y^{k-1}_{ij} \& Y^{k-1}_{ik} \cdot Y^{k-1}_{kk} \cdot Y^{k-1}_{kj}$
10     end
11   until $Y^n = Y^0$;
12   $Y \leftarrow Y^n$
13 end
```

PC-1: the first path consistency algorithm

Montanari [14] gives an inductive proof for the correctness of PC-1. Another justification derives directly from the theorem above. To see that the algorithm halts observe that the & operation of line 9 has a monotonic effect on Y_{ij}. On the iteration of lines 4–10 that the algorithm halts on, $Y = Y^n = Y^0 = Y^k \, (1 \leqslant k \leqslant n)$ and so line 9 has had no effect at all: for all i, j, k, $Y_{ij} = Y_{ij} \& Y_{ik} \cdot Y_{kk} \cdot Y_{kj}$. All paths of length 2 (v_i, v_j, v_k) are consistent so Y is path consistent.

Parenthetically, Algorithm PC-1 should be compared to Algorithm 5.5 of Aho, Hopcroft and Ullman [1] which is a generalization of Warshall's [25] transitive closure algorithm and Floyd's [8] shortest path algorithm. Algorithm 5.5 needs only one iteration of the equivalent of lines 4–10 because they require that · be distributive over + (& in our case) whereas here there is no guarantee that composition is distributive over intersection of binary matrices.

PC-1 is correct but it consumes more time and space than it need. In pursuing this thought it is profitable to see that PC-1 is a generalization of AC-1. PC-1 essentially becomes AC-1 if we substitute

$$8\ Y_{ii}^{k} \leftarrow Y_{ii}^{k-1} \mathbin{\&} Y_{ik}^{k-1} \cdot Y_{kk}^{k-1} \cdot Y_{ki}^{k-1}$$

for lines 8–9 of PC-1. (See Section 7.3.1.2 if this is not clear.)

Pursuing the comparison with AC-1, we ask if it is necessary to keep $n+2$ copies of the network of relations: R, Y^0 and Y^k $(1 \leqslant k \leqslant n)$, each of which will be very large even for moderate n. R is clearly unnecessary. To avoid keeping Y^0 use a flag which is set to *true* when any Y_{ij} is changed. Line 9 requires that one use Y_{ik}^{k-1}, Y_{kk}^{k-1} and Y_{kj}^{k-1} even though one or more of the updated versions Y_{ik}^{k}, Y_{kk}^{k} and Y_{kj}^{k} may already have been computed. Clearly the only possible effect of using the updated versions of those relations is to speed convergence. The outcome is then that only a single copy of Y which is continually updated need be used.

Secondly, some computations predictably have a null effect (e.g., $Y_{kk}^{k} = Y_{kk}^{k-1}$) so need not be done. Third, since $Y_{ji} = Y_{ij}^{T}$ almost half the computation can be avoided.

But these improvements are matters of detail not substance. A substantial improvement can however be effected by pursuing further the analogy with AC-1. There we noted that whenever an arc was made consistent by deleting values from the node at its tail rather than require another complete iteration through the entire set of arcs one could specify just which arcs might be affected and put them on a queue either to be dealt with when the current set of arcs was exhausted (AC-2) or whenever was convenient (AC-3). Here we can see that we are considering the entire set of paths of arc length 2. If a path is not consistent we make it so by changing the necessary 1's to 0's in the binary matrix relating the two terminal nodes of the path. When we do so every path of length 2 that has as one of its component arcs the arc between the terminal nodes of the path just made consistent must be (re)checked for consistency. However, some of these paths may already be waiting in the queue to be considered. As in the case of arc consistency we define a procedure REVISE which checks a path of length 2 from node i through node k to node j for consistency. If it must be made consistent by modifying Y_{ij} REVISE returns *true* otherwise *false*.

```
procedure REVISE ($(i, k, j)$)
begin
  $Z \leftarrow Y_{ij} \mathbin{\&} Y_{ik} \cdot Y_{kk} \cdot Y_{kj}$
  if $Z = Y_{ij}$ then return false
    else $Y_{ij} \leftarrow Z$; return true
end
```

We also need a procedure RELATED PATHS((i, k, j)) that returns a set of length 2 paths that need to be REVISEd if REVISE((i, k, j)) returns *true*. Since $Y_{ij} = Y_{ji}^T$ we need only compute Y_{ij} if $i \leqslant j$ so RELATED PATHS has two cases to consider: (a) $i < j$ and (b) $i = j$.

(a) $i < j$. i and j are distinct nodes so we want the set of all paths of length 2 that have arc (i, j) or arc (j, i) as one of their arcs. Also, we want to exclude paths (i, j, j) and (i, i, j) because on both REVISE will predictably return *false*.

In this case, the set of paths to be returned is

$$\begin{aligned} S_a = {} & \{(i, j, m) \mid (i \leqslant m \leqslant n), (m \neq j)\} \\ & \cup \{(m, i, j) \mid (1 \leqslant m \leqslant j), (m \neq i)\} \\ & \cup \{(j, i, m) \mid j < m \leqslant n\} \\ & \cup \{(m, j, i) \mid 1 \leqslant m < i\} \end{aligned}$$

S_a has $2n-2$ members.

(b) $i = j$. In this case Y_{ii} has changed so every path of length 2 that uses i as its intermediate node must be checked with the exception of paths (i, i, i) and (k, i, k). The set of paths to be returned is $S_b = \{(p, i, m) \mid (1 \leqslant p \leqslant m), (1 \leqslant m \leqslant n), \neg(p = i = m), \neg(p = m = k)\}$ S_b has $n(n+1)/2-2$ members. The paths (i, i, i) and (k, i, k) are excluded because they would result in REVISE returning *false*.

Note that the exclusion of (k, i, k) from the set of paths related to (i, k, i) corresponds exactly to the exclusion of arc (m, k) when REVISE((k, m)) was predictably *false* there as REVISE((k, i, k)) is predictably *false* here.

Finally,

procedure RELATED PATHS((i, k, j)):
if $i < j$ *then return* S_a *else return* S_b

Now we have the components for a more efficient path consistency algorithm, PC-2.

```
1 begin
2   $Q \leftarrow \{(i, k, j) \mid (i \leqslant j), \neg(i = k = j)\}$
3   while $Q$ is not empty do
4     begin
5       select and delete a path $(i, k, j)$ from $Q$;
6       if REVISE($(i, k, j)$) then $Q \leftarrow Q \cup$ RELATED PATHS($(i, k, j)$)
7     end
8 end
```

PC-2: the second path consistency algorithm

The order of path selection from Q does not affect the outcome of the algorithm but it may affect its efficiency. In particular if Q is ordered on the value of k then the initial set of paths is processed in essentially the same order as in PC-1. If the relations are such that composition does distribute over intersection then we are guaranteed that the value of Y^n after the first iteration of PC-1 lines 4–10 will be its final value, Y: on the second iteration there will be no further change. (This is so because in that case the task is that of Aho, Hopcroft and Ullman's [1] Algorithm 5.5. See that reference for a precise specification of a set of conditions sufficient to ensure that only one iteration of PC-1 is necessary.) Thus, if Q is so ordered in PC-2 then only on the original set of paths in Q (of which there are $(n^3+n^2-n)/2$) will REVISE return *true* and hence possibly increase the length of Q. Any other ordering may not have that effect.

8. The Use of Consistency Methods in Problem Solving

The consistency methods discussed were initially motivated here by reference to three situations that caused pathological thrashing behavior in a backtracking problem solver. How then are these consistency algorithms to be used? Clearly, applying PC-2 before backtracking will ensure that none of the thrashing behaviors discussed in Section 3 will occur; however, it is possible to do better. As Fikes showed in REF-ARF alternating constraint manipulation and instantiation of a variable is a good strategy for Boolean constraint problems. Burstall [3] in a program for solving cryptarithmetic puzzles alternated constraint manipulation and the bisection of variable domains. A formulation that includes these two approaches as special cases is the alternation of constraint manipulation and case analysis. By case analysis is meant the creation of p subproblems by adding to each of p copies of the network an additional case constraint where the p case constraints OR'ed together constitute a tautology. (The additional tautological constraint may involve more than one variable.) The resultant OR graph may be searched in any of the usual ways [16]; a solved subproblem has a unique instantiation of the variables after PC-2 has been applied (i.e., each Y_{ii} has exactly one 1 on the diagonal) whereas an unsolvable subproblem has some Y_{ij} with all entries 0 (in fact in that case all Y_{ij} will have all entries zero after PC-2 has been applied.)

9. Applications

9.1. Finite, discrete state space problems

9.1.1. *Puzzles*

The most obvious applications of these techniques are to the traditional puzzle-solving problems. Gaschnig [10], for example, has used an iteration of a modification of AC-3 and instantiation to solve Instant Insanity and cryptarithmetic puzzles and has shown how the search space is drastically reduced. That version of AC-3 does not, however, distinguish between the arc (i,j) and the arc (j, i) so that the equivalent of REVISE((i,j)) must check every value of D_i and find a corresponding value in D_j and also must similarly check every value of D_j for the existence of a compatible value in D_i, although as shown in Section 6 when REVISE is called on a pair of adjacent nodes it is known which of the two arcs is possibly inconsistent.

Other puzzles to which these methods apply are magic square problems and the n-queens problem. The list could be longer.

9.1.2. *Other combinatorial problems*

It remains to be seen whether the approach suggested will lead to more effective algorithms for such traditional combinatorial tasks as computing the chromatic number of a graph and the graph isomorphism problem although it is clear that Unger's [23] approach to graph isomorphism contains some of the seeds of this approach. In a similar vein Suzman and Barrow [21] have been applying an arc consistency algorithm to clique detection.

9.2. Continuous variable domains

The requirement that the relations between variables be explicitly represented does not lead of necessity to the Boolean matrix representation. As Montanari points out, any representation of the relations that allow composition and intersection is sufficient. For example, using as the domains subsets of R^n allows one to treat space planning [5] and n-dimensional space packing problems such as cloth cutting [11] and the FINDSPACE problem [20].

9.3. Vision

In the Waltz filtering algorithm the variables are picture junctions whose values are their possible interpretations as corners. The initial variable domains arise from the shape of the junctions; the unary predicates arise from lighting inferences while the binary predicates simply require each edge to have the same interpretation at both of its ends. An interesting question to pursue is to ask how the processing time depends on the complexity of the picture. From the available results [24], the dependence could well be linear. Waltz suggests that this is so because when each new junction is introduced the propagation of arc revision is restricted for the most part to that set of lines forming the image of a single body of which that junction is a part. The effect is so restricted because T junctions do not transmit constraining action from the stem to the crossbar or vice versa. Unless this decoupling effect obtains in other domains there is no reason to expect linear behavior from AC-2 or AC-3. Moreover, in this domain, the interpretation of pictures of more and more complex individual polyhedra rather than of more and more polyhedra of fixed complexity would not presumably display linear behavior. Worst case analysis of AC-1 suggests that the processing time is $O(a^2)$ where a is the number of arcs in in the graph. Turner [22] has generalized the Waltz' algorithm to apply to certain curved objects.

The author has previously proposed [12] the use of arc consistency algorithms in the task of interpreting pictures of polyhedral scenes. In that application, the variables are the regions whose values are the positions and orientations of their possible interpretations as surfaces. The unary and binary predicates arise both from

(a) Constraints on the surface positions and orientations taken individually and pair-wise together, if they intersect in an edge, imposed by the geometry of the

picture formation process. (These are the constraints exploited by the author's earlier program, POLY [13].)
And

(b) Constraints on surface size, shape and pairwise connectivity imposed by a priori knowledge of the objects that can appear in the world.

Barrow and Tenenbaum [17] have an application of arc consistency in which the variables are picture regions and the values are the names of their interpretations as surfaces (such as, "door", "wall" and "picture"). Rather than just satisfy the constraints they seek an assignment of values to variables that will maximize the likelihood of the region interpretations being correct. In a related study, Rosenfeld, Hummel and Zucker [18] investigate various probabilistic models using AC-1.

Finally, Montanari [14] suggested that the variables be distinctive, recognizable subpictures of the picture one is interpreting. If the values are the pictorial location of these subpictures then one could use the consistency algorithms and subpictures already located to constrain the search area for an as yet unlocated subpicture.

9.4. AI programming languages

Consistency methods could well solve some of the problems of retrieval from a data base that essentially takes the form of a semantic network. The criticisms that Sussman and McDermott [19] leveled at the crucial position occupied by automatic backtrack control in PLANNER were well founded and yet it is also clear that, unless we are to abandon completely the goal of a high-level programming language for AI, default search and data base retrieval mechanisms should be available to the user. (And yet again, these should not be forced upon the user. If he wants to program his own, the primitives should be available as they are in Conniver.)

The consistency methods advocated here are clearly more effective than automatic backtracking and so deserve to be considered as a default database retrieval mechanism.

As an example, "Find a large rectangle which is touching a triangle and inside a circle" could appear in MICRO-PLANNER as

```
(THPROG (X Y Z)
   (THGOAL (OBJ $?X RECTANGLE))
   (THGOAL (SIZE $?X BIG))
   (THGOAL (TOUCHING $?Y $?X))
   (THGOAL (OBJ $?Y TRIANGLE))
   (THGOAL (OBJ $?Z CIRCLE))
   (THGOAL (INSIDE $?X $?Z))
   (THRETURN $?X))
```

or in network form as in Fig. 5. One need not enumerate again all the thrashing problems that the execution of that MICRO-PLANNER code would encounter in various configurations of the world. As a simple example, just consider a situation in which there are a large number of rectangles only one of which is inside the only circle. Backtracking could thrash for a very long time before discovering the right rectangle whereas a "truly smart" procedure would take the circle, find out what is inside it, ... The effect of that smart procedure would be achieved by the consistency algorithms described here.

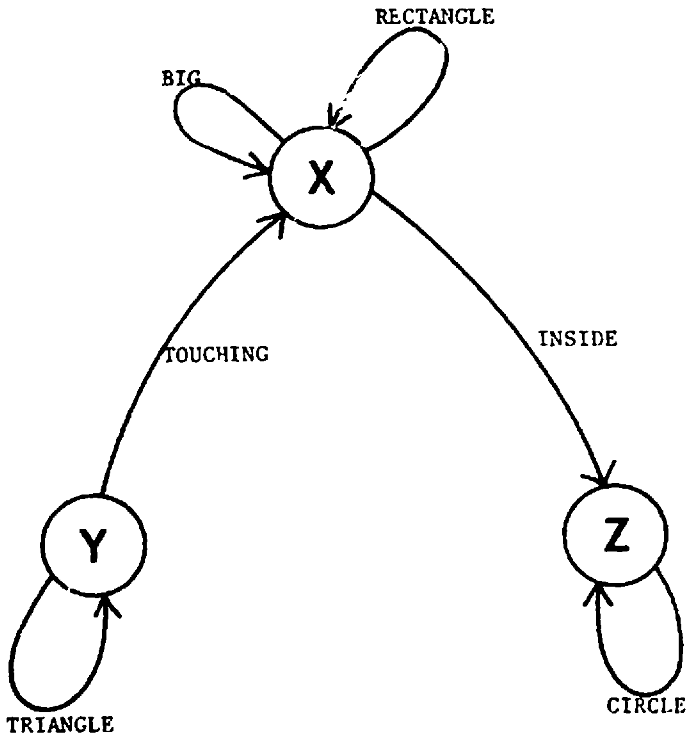


FIG. 5. A network representation of a retrieval task.

10. Conclusion

In this paper, we have been concerned with a class of algorithm, which could be named network consistency algorithms, designed to aid in the discovery of a situation that satisfies a set of simultaneous constraints that has been imposed on any candidate solution.

By being presented and extended in a uniform framework these algorithms will perhaps become more accessible to others as will the pursuit of their development in the context of a variety of applications many of which have been discussed here.

ACKNOWLEDGMENTS

I am indebted to F. O'Gorman for some initial conversations on this topic. The support of the National Research Council of Canada, under Grant A9281, is gratefully acknowledged.

REFERENCES

1. Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

2. Bobrow, D. G. and Raphael, B., New programming languages for AI research, *Comput. Surv.* 6 (1974), 153–174.
3. Burstall, R. M., A program for solving word sum puzzles, *Comp. J.* 12 (1969) 48–51.
4. Dahl, O. J., Djikstra, E. W. and Hoare, C. A. R. *Structured Programming*, Academic Press, 1972.
5. Eastman, C. M. Automated space planning, *Artificial Intelligence*, 4 (1973), 41–64.
6. Fikes, R. E. REF-ARF: A system for solving problems stated as procedures, *Artificial Intelligence*, 1 (1970), 27–120.
7. Floyd. R. W., Nondeterministic algorithms, *J. Assoc. Comput. Mach.* 14 (1967), 636–644.
8. Floyd, R. W. Algorithm 97: shortest path, *Comm. ACM* 5 (1962), 345.
9. Garey, M. R., Johnson, D. S. and Stockmeyer, L. Some simplified NP-complete problems. *Proc. 6th Annu. ACM Symp. Theory Comput.*, Seattle, Wash. (1974), pp. 47–63.
10. Gaschnig, J. A. Constraint satisfaction method for inference making, *Proc. 12th Annu. Allerton Conf. Circuit System Theory*, U. Ill., Urbana-Champaign (1974)
11. Haims, M., On the optimum two-dimensional allocation problem, Ph.D. Thesis, Dept. of Electrical Engineering, New York University, New York (1966).
12. Mackworth, A. K. Using models to see. *Proc. Artificial Intelligence and the Simulation of Behaviour Summer Conf.*, University of Sussex (1974), pp. 127–137.
13. Mackworth, A. K. Interpreting pictures of polyhedral scenes, *Artificial Intelligence*, 4 (1973), 121–137.
14. Montanari, U. Networks of constraints: fundamental properties and applications to picture processing, *Inform. Sci.* 7 (1974), 95–132.
15. Montanari, U. Optimization methods in image processing. *Proc. IFIP Congress*, North-Holland, 1974, pp. 727–732.
16. Nilsson, N. J. *Problem-solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.
17. Nilsson, N. (Ed.), Artificial intelligence—research and applications, progress report, Stanford Research Institute (1975).
18. Rosenfeld, A., Hummel, A. and Zucker, S. W. Scene labelling by relaxation operations, Computer Science TR-379, University of Maryland (1975).
19. Sussman, G. J. and McDermott, D. V., Why conniving is better than planning, Artificial Intelligence Memo. No. 255A, MIT (1972).
20. Sussman, G. J. The FINDSPACE problem, Artificial Intelligence Memo. No. 286, MIT (1973).
21. Suzman, P. and Barrow, H. G. Private communication, 1975.
22. Turner, K. J. Computer perception of curved objects using a television camera, Ph.D. Thesis, Dept. of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh (1974).
23. Unger, S. H., GIT—a heuristic program for testing pairs of directed line graphs for isomorphism, *Comm. ACM*, 7 (1964), 26–34.
24. Waltz, D. L., Generating semantic descriptions from drawings of scenes with shadows, MAC AI-TR-271, MIT (1972).
25. Warshall, S. A theorem on Boolean matrices, *J. Assoc. Comput. Mach.* 9 (1962), 11-12.

Received January 1976; revised version received June 1976