[19] Consistency Maintenance in the REVgraph Environment

Jonathan B Bowen and John E W Mayhew

AI Vision Research Unit
University of Sheffield, Sheffield S10 2TN, UK

Reprinted, with permission of Butterworth Scientific Ltd, from *Image and Vision Computing*, 1988, **6**, 12-15.

Introduction

The REVgraph is a data structure designed to represent a 3 dimensional model of a scene, built up from a stereo pair of 2 dimensional images. The most important levels of representation are Regions, Edges and Vertices, linked together as a graph structure, though other intermediate structures also exist.

The REVgraph is built up from our stereo processing system, TINA[1]. This system runs a Canny edge detector over both images, finds disparity values for the edge points using the PMF stereo correspondence algorithm[2] groups up the points into edge segments, and produces a geometric description of these edge segments in terms of straight lines and circular arcs[3]. The REVgraph contains representations for the outputs of all these primary stages of visual processing.

The geometric descriptions of the edge segments are typically fragmented due to noise and to the intrinsic nature of the edge detecting processes which tend to break up edges at, for example, vertices.

The use of the REVgraph is to enable reasoning processes to reconstruct a geometrically consistent description of the surfaces in the scene from the various lower levels of description. This will involve completion of broken edges, the finding of vertices and finally the identification and description of regions bounded by edges and vertices.

For the purposes of the reasoning process, any piece of data in the REVgraph may be regarded an assertion or fact. The nature of such facts will depend on the level of the data : At the edge detection level, a fact will be the presence of an intensity gradient maximum at a particular pixel. At the geometric level, it will be the direction and end points of a straight edge segment, and at higher levels the existence of a vertex or region, or ultimately the identification of a group of regions as a recognised object.

The reasoning processes required for this task utilise a set of rules that express knowledge about the task. For example, a typical rule is : "If 3 lines can be extended to meet at point Then hypothesise a vertex at that point." It is frequently a matter of considerable uncertainty as to which rules should be applied at any one time due to ambiguity in the initial data, so we require a method of exploring several alternative lines of reasoning with the ability to recover gracefully from contradictions and errors.

To this end, we have been exploring Truth Maintenance Systems (TMS) as a method for controlling the reasoning system. We have studied three particular Truth Maintenance algorithms in detail : Doyle's[4], De Kleer's[5] and McDermott's[6]. In all three we have found limitations which make them inappropriate for our domain, but all have some very attractive properties which we have tried to combine.

Truth Maintenance is concerned with taking a database of facts (or assertions) some of which may be contradictory, and, using the paths of justifications for the facts, partitioning the database into one or more self-consistent sets of facts. Such a set of self-consistent facts we will in future refer to as a solution to the truth maintenance problem. Some truth maintenance systems only follow one solution explicitly, while others follow many competing possibilities at once. Also, some only find the solution(s) when all deductive processing has been completed, while others represent explicit part-solutions at every stage through reasoning.

Existing Truth Maintenance Systems

Doyle's Truth Maintenance System was evolved for non-monotonic reasoning, where the belief of more assumptions does not necessarily lead to belief in more facts, but may result in fewer facts being believed. Thus the database of True or believed facts does not necessarily grow as further deductions are made. The system was designed to yield a single consistent set of facts as a solution. If this solution is rejected, by the discovery of a contradiction or by user intervention, the system then backtracks, at considerable expense, to find another solution.

McDermott's TMS is an attempt to reconcile Doyle's ideas with some earlier ideas of contexts. The main extension to Doyle's system is the facility for user programs to label major decision points, and for the database at that stage to be "pushed down", for easy recovery later. This would allow a certain reduction in the thrashing behaviour of TMS if such pushed down contexts were requested to be re-instated after a contradiction.

De Kleer's Assumption-Based Truth Maintenance System (ATMS) arose with the need to represent several solutions, one of which could then be chosen. Also, his algorithms are highly optimised. Part of this increase in efficiency is achieved by abandoning Doyle's non-

monotonic justifications (eg If x=1 is true, then y=0 is NOT true) in favour of representing known contradictions in the database of facts explicitly. In the basic ATMS this places some restriction on the domains that the system is useful within, but in an extended version such restrictions are removed with the penalty of reduced efficiency.

Reasoning Within the REVgraph

The style of reasoning we will be using with the REV-graph is relatively straightforward. It involves a set of rules which will search for certain patterns of data in the REV-graph (antecedent data), and from this will generate new data at a different level in the graph. From here on the antecedent facts are referred to as the premises of the deduction, and the new data is referred to as the consequent of the deduction.

Due to the ambiguity in the initial data, some rules may hypothesise more than one possible consequent, and some deductions may result in geometric inconsistencies. We intend to use truth maintenance techniques to keep track of consistent solutions, and to aid decisions on which facts and deductions to disbelieve.

The following behaviours would be required of any TMS :

1) Work done towards one solution should be automatically inherited by other possible solutions wherever it is relevant.

2) Different justifications for the same fact should not become confused.

3) When a contradiction is found, the premises that gave rise to that contradiction should never be allowed to co-exist within a solution, nor should they be allowed to generate any new facts in the database.

4) A fact should not be allowed in a solution if all its paths of deduction include itself. This is known as Circular deduction and the problem is somewhat similar in nature and complexity to garbage collection of circular data structures.

5) It should be possible for the user to intervene in processing to arbitrarily add facts to or remove facts from the database.

6) It would be desirable to provide facilities for the problem solver to be able to intelligently bound the potentially very large search space. The burden of the intelligence, of course, lies with the problem solver.

There are three properties we consider to be of greatest importance for a TMS in the REVgraph :

1) At any time it must be possible to traverse a self-consistent, if incomplete, graph, as further reasoning and local inconsistency discovery can only be done by traversing the graph. This is difficult with De Kleer's scheme, as there is no notion during processing of a part solution. De Kleer states[7] "It is extremely rare for a problem solver to ask for the contents of a context." and uses this assumption to increase ATMS efficiency. We are particularly interested in cases where this assumption breaks down.

2) The initial data is ambiguous rather than wrong. Consequently the rules used for reasoning will be based largely on heuristics rather than on logical implications. Thus, in a backtracking scheme, if a fact is found to be wrong, there may be no reason to believe that its premises are wrong. It may simply be that the heuristic that generated the fact got it wrong on this occasion. None of the schemes we investigated provide what we consider to be an appropriate method for dealing with this situation.

3) Often, ambiguities may lead to radically different solutions of the graph. Rather than find one solution and then backtrack to find others, it would be more efficient to follow the development of alternative solutions in parallel, and compare final solutions to choose the "best". Doyle's TMS does not allow this, but De Kleer's ATMS was developed primarily to give this property as efficiently as possible.

The Proposed Solution : *Consistency Maintenance*

Since our reasoning system will be using heuristics rather than logically correct rules, there is less an element of solutions being True, and more of them simply being Consistent. Hence we have called our proposed system a 'Consistency Maintenance System' (CMS).

The key idea in CMS is that of the 'Context': The database of facts will be interconnected by logical dependencies, such as "Fact1 justifies Fact2, therefore Fact2 cannot exist without Fact1". Also, some of the facts may be contradictory, and therefore cannot exist in the same solution. It is possible to divide the database into subsets within which there are no contradictory facts, and no paths of justification are incomplete. Such a subset we refer to as a 'Context'.

eg Suppose :

Fact1 and Fact2 justify Fact3.
Fact4 and Fact5 justify Fact6.
Fact6 and Fact7 justify Fact8.
Fact3 and Fact6 justify Fact9.
Fact10 doesn't justify anything else.

AND Fact2 and Fact4 are known to be true,
AND Fact3 is then found to be contradictory to Fact6.

The resulting database could be broken down as in Figure 1. The proposed representation of this division of the database into contexts is :

Fact1	(1)
Fact2	(1 2)
Fact3	(1)
Fact4	(1 2)
Fact5	(2)
Fact6	(2)
Fact7	(1 2)
Fact8	(2)
Fact9	()
Fact10	(1 2)

Hence :

Context 1	Context 2
Fact1	Fact2
Fact2	Fact4
Fact3	Fact5
Fact4	Fact6
Fact7	Fact7
Fact10	Fact8
	Fact10

These contexts are consistent partitions of the global database.

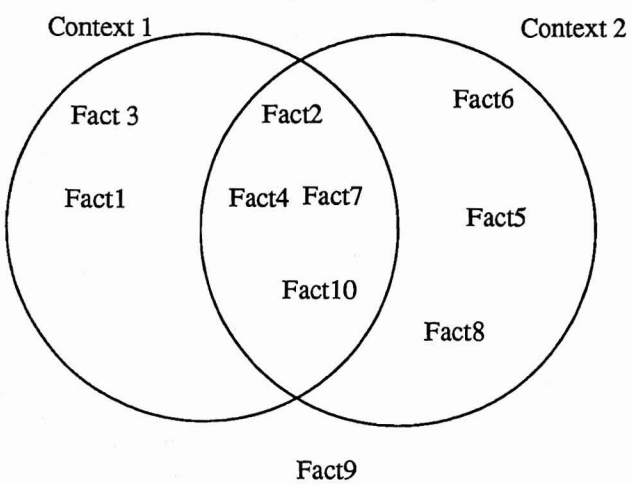


Figure 1 :- Division of Database after a contradiction

Inheritance of work between contexts

Suppose in the above example 3 more deductions were made :-

Fact1 and Fact7 justify Fact11.
Fact5 and Fact10 justify Fact12.
Fact7 and Fact10 justify Fact13.

The contexts of Fact11, Fact12 and Fact13 can be calculated from their justifications as shown in Figure 2.

Representation :

Fact1	(1)
Fact2	(1 2)
Fact3	(1)
Fact4	(1 2)
Fact5	(2)
Fact6	(2)
Fact7	(1 2)
Fact8	(2)
Fact9	()
Fact10	(1 2)
Fact11	(1)
Fact12	(2)
Fact13	(1 2)

Hence :

Context 1	Context 2
Fact1	Fact2
Fact2	Fact4
Fact3	Fact5
Fact4	Fact6
Fact7	Fact7
Fact10	Fact8
Fact11	Fact10
Fact13	Fact12
	Fact13

The rule governing inheritance is as follows :

The contexts in which a new fact is valid are the intersection of the contexts in which its premises are valid.

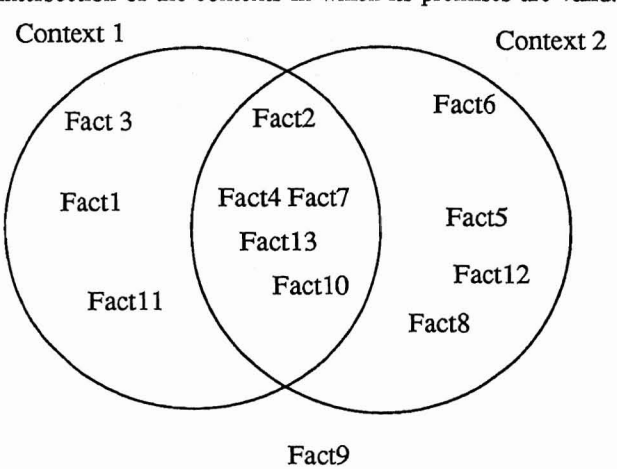


Figure 2 :- Division of Database after further deductions

Multiple Justification of Facts.

Suppose in this example that one further deduction were made :

Fact4 and Fact1 justify Fact12.

Fact12 already exists in context 2, justified by fact4 and fact10. However, the new justification makes it valid in context 1. As this does not lead to any contradictions, fact12 should now become valid in both contexts. See Figure 3.

Representation :

Fact1	(1)
Fact2	(1 2)
Fact3	(1)
Fact4	(1 2)
Fact5	(2)
Fact6	(2)
Fact7	(1 2)
Fact8	(2)
Fact9	()
Fact10	(1 2)
Fact11	(1)
Fact12	(1 2)
Fact13	(1 2)

Hence :

Context 1	Context 2
Fact1	Fact2
Fact2	Fact4
Fact3	Fact5
Fact4	Fact6
Fact7	Fact7
Fact10	Fact8
Fact11	Fact10
Fact12	Fact12
Fact13	Fact13

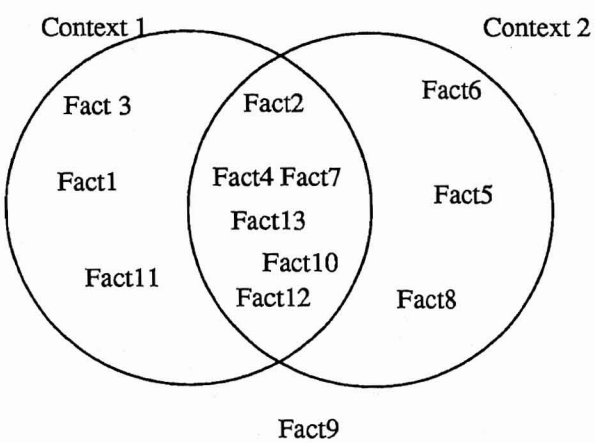


Figure 3 :- Division of Database with a multiple justification

The generalisation of the rule above is this : Each justification for a fact is valid in a set of contexts which is the intersection of the contexts in which the premises are valid. The fact itself is valid in a set of contexts which is the union of those contexts in which its justifications are valid.

The Data Dependency representation.

As processing of the REVgraph progresses, we need to record the deductions that have taken place. Following Doyle, the data structure we have decided upon is a directed graph consisting of two types of node, the 'Fact node' and the 'Data Dependency node'.

Data Dependency Node (DD-node)

The DD-node represents a single deduction by the reasoning system.

Premises List	--> List of fact nodes which are required to make this deduction. empty if the consequence is an axiom
Consequence	--> Pointer to a single fact node, which is the result of the deduction.
Rule Identifier	--> Pointer into the rulebase, for recording information for the problem solver's use
Context List	--> List of contexts in which this justification is valid. It is the intersection of the contexts of the premises, pointed to above.
Disallowed Contexts	--> Contexts in which this deduction is invalid due to a contradiction

Fact Node

The fact node represents an actual item in the database.

Support-By List	--> List of all the DD-nodes which form justifications for this fact.
Support-For List	--> List of all the DD-nodes for which this fact is a premise.
Value	--> A pointer into the database to the actual value which this fact node represents.
Context List	--> A list of the contexts of which this fact is a member. This is the union of the contexts of the Support-By DD-nodes.
Contradictions	--> A list of lists of fact nodes. Each 2nd level list represents one contradiction of which this fact is a part.

Constraints Between Facts in the Database

As we mentioned above, the development of the REVgraph will be guided by heuristics as much as by logically correct rules. The constraints between the premises and consequence of a heuristic deduction are slightly different from those between the premises and consequence of a logically correct deduction. The following descriptions

apply to facts with only one justification. The generalisation to multiple justifications is given in the next section.

Suppose Fact1, Fact2 and Fact3 together are evidence for a heuristic rule to propose the existence of Fact4. Let us refer to the premise set as P, and the Consequence as C. Then, a context within which C is valid (or 'True'), must also contain the evidence for C (i.e. P must be True as well). As an example, it would be ridiculous to propose the existence of a vertex at a point if no edges terminated at that point. However, if P is True in a context, then C may or may not be True in that context, because the result of the heuristic operating on P cannot be trusted to be definitely correct. This argument can be represented in a Boolean expression :

" P heuristically justifies C " or : (P ħ C)

The Boolean table for the operator ħ is :

P	ħ	C	P heuristically justifies C
T	T	T	A context where the evidence is True and the consequence is True is valid within the CMS.
T	F	T	The evidence is True, but the consequence is not trusted, and is made False.This is valid.
F	T	F	A context where the consequence is True despite having no evidence is NOT valid within CMS.
F	F	T	A context in which neither the evidence nor the consequence is True, is valid in the CMS.

So, for this deductive step on its own, only contexts which satisfy that constraint between the evidence and consequence, may exist. The job of the CMS is to maintain those constraints properly, and never generate a context where the constraints are not satisfied, such as a context which contains C but does not contain P.

It may be noticed that for any heuristic deduction there are two possible contexts in which the premise set is true. Rather than keep track of both contexts from the outset, the CMS first follows the context that contains the consequence, but if this causes a contradiction it then falls back to the context in which the premise is true but the consequence is not. This highlights an important property of the CMS, that of maximality of the contexts represented : That is, for any context, the addition of further facts from the database to that context would necessitate a contradiction.

Logically correct, or rigorous rules : This is the kind of rule of that Truth Maintenance systems have conventionally dealt with, where if the premises are True, the consequence is undeniably True . For example, if the endpoints of an edge are defined, then the direction must be the vector difference between them. Again, however, the consequence cannot exist without the evidence. Thus, if the end points of an edge are undefined, it would be silly to give a value for the direction. This can also be represented as a Boolean expression.

Defining P and C as above, let us see the effect of this statement :

"P rigorously justifies C" or : (P r̃ C)

The Boolean table for the operator r̃ is :

P	r̃	C	P rigorously justifies C
T	T	T	A context where the evidence is True and the consequence is True is valid within the CMS.
T	F	F	A context where P is True and P rigorously justifies C, but C is False, is not valid in CMS.
F	T	F	A context where the consequence is True despite having no evidence is NOT valid within CMS.
F	F	T	A context in which neither the evidence nor the consequence is True, is valid in the CMS.

These are the constraints that the CMS must maintain between the premises and consequence of a logically correct rule. They are a little more restrictive, having fewer valid states than the heuristic implication.

Other constraints that exist within the database :

Suppose Fact1 and Fact2 and Fact3 are axioms (unjustified facts) which the user wishes to believe in, then the constraint that :

(Fact1 ∧ Fact2 ∧ Fact3) is True must hold in all contexts. i.e all contexts must contain these facts. Axioms that the user doesn't trust don't impose any constraints on contexts.

Axioms in the CMS are, in fact, represented as either heuristic or rigorous deductions. An axiom that the user trusts is represented as a rigorous deduction from "True", while untrustworthy axioms, or assumptions, are represented as heuristic deductions from "True". The treatment of heuristic deductions as explained above means that untrustworthy axioms are believed until they are found to cause a contradiction.

Also, any time a contradiction is found, an extra constraint is placed on the database : Suppose the combination of Fact1, Fact2 and Fact3 is found to be contradictory, then the constraint that ¬(Fact1 ∧ Fact2 ∧ Fact3) must hold in all contexts. i.e. no context may contain all the facts involved in a contradiction.

Generalising the operators ħ and r̃, if a fact has several justifications of different types, the situation becomes more complicated. If P1, P2 and P3 are 3 premise sets each of which provides a rigorous justification for Fact1, and P4 and P5 are premise sets each of which provides a heuristic justification for Fact1, then the following constraint must be True :

((Fact1 ⇒ (P1 ∨ P2 ∨ P3 ∨ P4 ∨ P5)) ∧ ((P1 ∨ P2 ∨ P3) ⇒ Fact1))

Where each of P1 -> P5 is of the general form

(Factm ∧ Factn ∧ ..... FactN).

This embodies the information that if a fact is True then at least one of its justifications is True, and also if any rigorous justification of a fact is True then that fact must be True. As described earlier, a justification is True if and only if all the facts in its premise set are True.

In this way, the function of the CMS may be seen as building a Boolean expression defining the constraints between facts in the data base, under the guidance of the reasoning system, and then solving the expression for its True values to give all the allowable contexts. However, we have not built it like this for the following reasons :

1) There are an enormous number of possible solutions to a large and complicated set of constraints such as those described, when the number of facts in the database becomes large enough to be worth dealing with. It is possible to keep track of a few solutions, and when a constraint is violated, to change one solution into alternatives which were not previously interesting, and hence not previously explicitly represented.

2) We do not build a Boolean expression as the database is built up, but instead we build the DD network, which is equivalent in the information contained, but far more convenient for maintaining the constraints.

The maintenance of the constraints of justifications is conceptually straightforward. Whenever the contexts in which a fact is valid are changed, either as the result of a new justification or a contradiction, then the contexts of all the justifications that fact is involved in are changed, and the contexts of the consequences of those justifications, according to the set operations described above. These changes are recursively fed forward through the DD network from premises to consequences until no further changes occur.

There are two complications to this algorithm. The first is that a context must not be fed through a heuristic deduction that has already been disallowed in that context, or any of its ancestral contexts. This can only occur when new justifications are being made. The second is that if contexts are being removed from facts, they might not be removed properly from facts involved in circular paths of deduction. This can only happen when a contradiction is being resolved. Special mechanisms are incorporated into the feed forward procedures to deal with these situations.

The SPLURGE Contradiction algorithm.

Imagine that the database consists of facts labelled ' A , B , C ..... , Z , ...'

Now suppose that the reasoning system discovered that facts A , B , C , D and E are contradictory facts. They must be split up in some way so that they don't all occur together in any context. Using the terminology above, a new constraint is added to the fact base, that of ¬(A B C D E). Those contexts which violate that constraint must be examined so that new contexts which do not violate the constraint may be found, and the old one(s) destroyed.

Contexts are allowable with all the rest of the facts (F , G , ... Z , ...) in the context, but with the following combinations of those facts involved in the contradiction :

A B C D	A B C E	A B D E	A C D E	B C D E
A B C	A B D	A B E	A C D	A C E
A D E	B C D	B C E	B D E	C D E
A B	A C	A D	A E	B C
B D	B E	C D	C E	D E
A	B	C	D	E

This is the 'Complete Splurge'.

This generation of new contexts seems excessive. $(2 \uparrow n) - 2$ contexts where n is the number of facts involved in the contradiction. We do not actually need to generate all these contexts, rather we have to be sure that the smaller contexts are available from the larger ones if the larger ones later become inconsistent themselves. This we can be sure of if we follow up the top line of the above list :

A B C D A B C E A B D E A C D E B C D E

This is the 'Maximal Splurge'.

This gives rise to n new contexts, where n is the number of facts involved in the contradiction. By repeated application of the maximal splurge algorithm to the new contexts, one can regenerate all the smaller contexts in the complete splurge. If any of the above contexts were not followed up, an avenue of exploration would be cut off from the CMS. This is basically how we follow only interesting contexts, but when a contradiction is found we are able to generate other contexts not previously explicitly represented.

Now, suppose that we had prior knowledge that facts A and B were actually trustworthy, but the rest were not. Then only the following contexts need be followed:

A B C D A B C E A B D E

This is the 'Smart Splurge'.

This gives even fewer contexts, and after a few contradictions the effect of simply knowing a couple of facts are trustworthy enables the search through contexts to be significantly reduced.

This example would be very simple if all the facts in the contradiction had exactly one heuristic justification each. However, in the general case, each fact will have more than one justification, and each such justification may be either heuristic or rigorous. Worse still, the different justifications may well be valid in different contexts, some of which contain the contradiction, and some of which don't. Determining what new contexts to create, and which facts should be valid in which context is quite complicated, but we here try to present a coherent outline of the necessary procedures.

The first task is to identify all those contexts in which the contradiction occurs. This is simply the set intersection of the contexts of the offending facts. For simplicity, we describe a version of the algorithm that treats each such context in turn, eliminating the contradiction from one context after another until all contexts are once again consistent. In practice, this approach is inefficient, but the actual algorithm implemented is essentially the same.

For each inconsistent context it is necessary to find the axiomatic justifications of each fact involved in the contradiction. An axiomatic justification for a justified fact is a minimal set of axiomatic facts from which the justified fact can be derived. Any fact may have several such axiomatic justifications. Finding these justifications involves traversing the DD network back through deductions until EITHER the item "True" is arrived at, OR a heuristic deduction is found. The axiomatic justification of a fact can then be recursively computed as follows : The axiomatic justifications of a fact are simply the set union of the axiomatic justifications of that fact's justifications. One axiomatic justification of a DD node is obtained by taking one axiomatic justification from each premise and unioning the elements together. All the axiomatic justifications are found by doing this with all the possible combinations.

This process is very similar to label generation in De Kleer's ATMS[7], but with important differences. First, a heuristic deduction is its own label, irrespective of the labels of its premises. Second, the labels need not be checked for consistency and minimality until the end.

When all the axiomatic justifications of each fact have been found, they can be checked for consistency and minimality. Basically this means that any which are supersets of others can at this point be discarded. The importance of these lists of axioms and heuristic deductions is that if a context existed which did not contain precisely one fact from each axiomatic justification, then that context would also not contain one of the facts involved in the contradiction. Thus, that context would become consistent. Only heuristic facts can be removed from a context, so all axioms which are rigorous deductions from "True" can be ignored from now on.

The axiomatic justifications for each fact are now turned into lists of out lists. One out list is formed by picking one (heuristic) fact from each axiomatic justification generated for one of the contradictory facts. This is done for all possible combinations, and for each of the contradictory facts. Out lists which are supersets of other out lists can now be discarded.

New contexts are created from the one under consideration by creating a new context label for each out list. Everywhere the original context appears in the database, it is replaced with this new set of context labels. Then, for each out list, every fact in that list is disallowed from that out list's corresponding context, and the resultant changes are fed forward through the DD network. Actually, it is heuristic deductions that are disallowed rather than facts, but this would have been too confusing to explain from the beginning!

In theory this process is repeated for every context in which the contradiction occurs, resulting in a new set of maximal and consistent contexts.

Optimisation of Context Lists

The representation of the contexts in which a fact is valid is very inefficient when a contradiction is found. When a context is split, it must be replaced everywhere it occurs in the DD network by the new contexts, which means that every fact node and every DD node in the database must be examined, and a set membership operation performed on every context list. It would be much better if only those nodes leading to or resulting from contradictory facts needed to be updated. We here present a representation of context lists that makes this the case.

An individual context will no longer be a single integer, but a list of integers. Each integer in this list represents a point where a contradiction was found and a context split into any number of new contexts. Thus, if the context labelled (1 3) were split into three new contexts, the new context labels would be (1 3 1) , (1 3 2) and (1 3 3). However, facts still valid in all three contexts can still be referenced by the single label (1 3). Thus, facts not in any way involved in the contradiction need not have their context lists adjusted at all. This representation of contexts can be seen as a tree. An example of such a tree relevant to the following discussion is shown in Figure 4. The original context representations are shown enclosed in square brackets, whereas the more optimal representations are shown enclosed by round brackets.

Under this new scheme, a context list that previously had the form:

{[5] [6] [7] [8] [9] [10]}

would now have the form

{(1 1)(1 2)(1 3 1)}

where the first three labels refer to original contexts 1 2 and 3, and the last label refers to both the original contexts

4 and 5. Thus, the context list is now a list of special labels, rather than actual contexts.

Making sure that all the other algorithms carry over is very simple. All the operations on context lists are set operations, so in order that they carry across, it is only necessary to define set operations on context lists using the new labelling scheme.

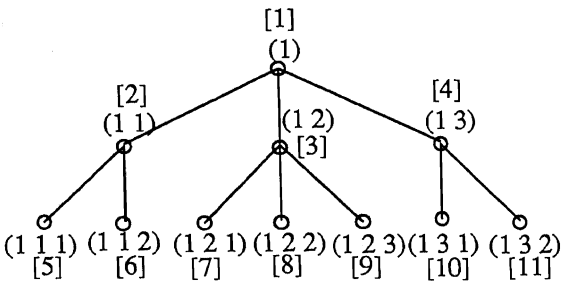


Figure 4 :- Context Tree with 2 context representations

Set Membership

To Compute Set Membership, the shorter label must be the head of the longer label :

(1 2 3) ∈ {(1 2 2)} is False

Previously :

[9] ∈ {[8]} is False

(1 2 3) ∈ {(1 2)} is True

Previously :

[9] ∈ {[7] [8] [9]} is True

Set Union

{(1 2)} ∪ {(1 3 1)} = {(1 2)(1 3 1)}

{(1 2)} ∪ {(1 2 1)} = {(1 2)}

{(1 2) (1 1 2) (1 3 1)} ∪
{(1 2 1) (1 1) (1 3 2)} =
{(1 2) (1 1) (1 3 1) (1 3 2)}

Previously :

{[7] [8] [9] [6] [10]} ∪
{[7] [5] [6] [11]} =
{[5] [6] [7] [8] [9] [10] [11]}

Set Intersection

{(1 2)} ∩ {(1 3 1)} = { ∅ }

{(1 2)} ∩ {(1 2 1)} = {(1 2 1)}

{(1 2) (1 1 2) (1 3 1)} ∩ {(1 2 1) (1 1) (1 3 2)} = {(1 2 1) (1 1 2)}

Previously :

{[7] [8] [9] [6] [10]} ∩ {[7] [5] [6] [11]} = {[7] [6]}

Set Difference

{(1 2)} − {(1 3 1)} = {(1 2)}

Previously :

{[7] [8] [9]} − {[10]} = {[7] [8] [9]}

{(1 2)} − {(1 2 1)} = {(1 2 2)
(1 2 3)}

Previously :

{[7] [8] [9]} − {[7]} = {[8] [9]}

The set difference operator requires more information than is available in the context list. To calculate the last example, reference must be made to the context tree, which is developed as contradictions occur. The set difference operator is only used at one point in the algorithms so far described, and that is when a new context is being fed forward through a heuristic DD node. The resultant contexts of the DD node is the intersection of its premises' contexts, less the contexts on its disallowed contexts list.

Comparison of CMS with Other Systems

We shall now compare the behaviour of CMS with Doyle's TMS, McDermott's TMS and De Kleer's assumption based technique. To demonstrate how they are related, we will return to the idea that the constraints discovered to exist between facts can be represented as a Boolean expression, and that valid solutions to the problem are those where the expression is True. Let us take the situation where some processing has been done, but no contradictions have been discovered.

The Boolean expression may look something like this :

(A ∨ B ∨ ....) ∧ (((A ∧ B) ∨ ....) ⇔ C) ∧ .... ∧ (..... ⇔ D)							
T	T	T	T	T	T	T	<-TMS , CMS
T	T	T	T	F	.	F	
T	F	T	F	T	.	F	
T	F	T	F	F	.	T	
F	T	F	T	T	.	T	
F	T	F	T	F	.	T	
F	F	F	F	T	.	F	
F	F	F	F	F	.	F	
T	T	T	T	T	.	T	
.	.	.	.	.	.	F	
		etc.					

Up to this stage, TMS will have done no backtracking, and will be following the topmost solution. CMS will be in the same situation, whereas De Kleer will have kept track of the constraints without explicitly solving any of the truth values. Now consider what happens if a contradiction is found between A and B. This adds a new clause into the expression, $\neg(A \wedge B)$, which has this effect :

(A ∨ B ∨ ..) ∧ (((A ∧ B) ∨ ..) ⇔ C) ∧ .. ∧ (.. ⇔ D) ∧ ¬(A ∧ B)								
T	T	T	T	T	T	F	F	
T	T	T	T	F	.	F	F	
T	F	T	F	T	.	T	F	
T	F	T	F	F	.	T	T	<-TMS, CMS
F	T	F	T	T	.	T	T	<-CMS
F	T	F	T	F	.	T	T	<-CMS
F	F	F	F	T	.	T	F	
F	F	F	F	F	.	T	F	
T	T	T	T	T	.	T	T	
.	.	.	.	.	.	T	F	
		etc.						

The topmost solution has become invalid. TMS backtracks to the next highest level solution, and if, in future, this is invalidated, TMS will attempt to find the next solution down, and so on. CMS keeps track of the next several solutions, which are all the maximal contexts. De Kleer's method would set up a nogood set, but would still have the work of interpretation construction to do. Also, De Kleer's method would not be able to show alternative 'part-solutions' without doing that work, so that development of the various solutions could not be easily followed. However, the ATMS would be able to use that nogood to prevent inconsistent facts from being used together to form further deductions. McDermott's TMS will behave as Doyle's, except that in response to user requests it can also set aside other specific solutions which it can move to quickly.

The main advantages of CMS over both these systems for our problem is that it caters for both rigorous and heuristic types of rule, and it successfully keeps track of equally valid alternative solutions as processing proceeds.

There are two major disadvantages of the CMS. Although it is based on a simple idea, the machinery necessary to put it into practice is very cumbersome, and also, as the complexity of the problem increases, the performance is likely to degrade exponentially. De Kleer's methods of interpretation construction to some extent reduce this problem, but CMS is likely to grind almost to a halt as the number of contexts becomes large compared to the number of facts in the database. In our domain, the problem is well enough determined for this situation to be unlikely to occur, and we are also investigating methods of abandoning unpromising contexts during processing, thus pruning the search.

The CMS as described above has been implemented, and applied to a typical vision problem, one particular stage in the process of WireFrame Completion. A brief description of the problem, the algorithm used to solve it and the role of the CMS follows, together with results that demonstrate the existence of ambiguity to a limited extent in this particular vision problem.

Prototype WireFrame Completion (PWFC)

The 3 dimensional geometric data recovered from the low level vision suite is segmented and described in terms of straight lines and circular arcs by the segmentation algorithm. Also, individual straight lines and circles are broken up due to noise, so that much of the topology of the edges in the scene is missing at this level. The aim of the PWFC algorithm is to reconnect the geometric data to find vertices and T junctions where appropriate.

The first version of this algorithm only works on the straight lines in a scene, and was written while the mathematics for circular arc geometry was being investigated. Both these tasks are now completed and this algorithm will soon be extended to include circular arcs.

At this stage there is a certain amount of ambiguity, and it was proposed to use the CMS to aid in searching parallel possibilities. Also, all the processing up to this point has been bottom up in nature, but it should now be possible to utilise top down information where available to ease the wireframe completion task. The CMS is useful in combining bottom up information with top down directives.

When we state that an object is justified, or given a justification, what is meant is that in addition to creating that object and linking it to others in the PWFC data structures, a request is sent to the CMS giving the object, those objects which justify it and a tag relating to the point in the program that this request is made. Similarly, when we state that a contradiction is found, or that objects are contradictory, what we mean is that a request is sent to the CMS informing it that those objects in the PWFC data sturcture have been adjudged to be contradictory.

Information relating to which objects are valid which others (i.e. in the same context) is obtained by making various requests to the CMS. The CMS can basically make

available information answering these two questions : "What contexts are this object valid in?", "What objects are valid in this context?". Any further information the user may require can be phrased in terms of set operations and those two questions.

Top Down Directives

(1) Focus on certain edges. The algorithm will only attempt actively to form completions (into vertices or T junctions) for a set of edges present in the interesting-wires list. Thus, if the higher level processes only demand processing of part of the scene, this can be achieved.

(2) Focus on certain pairings. The algorithm will initially attempt to form vertices between pairs of edges present on the interesting-pairs list. However, if these pairings are inconsistent, or very dubious, the use of the CMS allows the algorithm to explore other more promising routes.

(3) Restrict connections to a certain proximity. A true breadth first search algorithm will form connections between all possible combinations of edges. To restrict the search, junctions are only investigated if they occur within a user specified distance of the edge being completed.

How these top down constraints are implemented, and how the CMS is used to constrain the search using these constraints will be discussed in detail later on.

Data Structures

Wires : These represent straight lines as passed from TINA. They are introduced to the CMS as trustworthy axioms. Thus, all straight lines from TINA will exist in all solutions.

Connectors : These are straight lines that connect wires to each other, wires to vertices or wires to T junctions. These basically fill in the gaps due to noise and segmentation. How they are justified depends on the nature of the junction that is created.

Vertices : These are junctions where two wires can be extended to within a certain distance of each other in 3 dimensions at their point of intersection on the image plane. The two wires will be connected to the vertex by one connector each.

T Junctions : These are junctions where two wires meet on the image plane, but are further apart in 3 dimensions than a certain threshold. The occluded wire is connected to the T junction by one connector. The occluding wire must either directly occlude the other, or have a connector which occludes the other.

Super-vertices : where more than one vertex occur close enough to be considered the same, a super-vertex is created which groups them together. In this way, 3 or 4 directional vertices can be built up from 2 way vertices without further explicit representation.

Interesting-wires : As previously mentioned, this is a list of wires where attention is to be focussed.

Interesting-pairs : Another attention focussing list. This consists of pairs of wires, which are joined together by the completion algorithm in preference to other possibilities.

Candidate-lists : Each wire end has associated with it all the possible junctions that can be made from that wire end, within the user defined limiting radius. Initialisation of these lists is explained below.

Initialisation

Before the main algorithm can be run, various data structures must be initialised.

All the wires must be created from the GDB, and introduced to the CMS. The user must then define the limiting radius (which may be increased later), and must set the interesting-wires list.

The candidate lists must be created for the end of each wire in the interesting-wires list. This utilises the Pairwise Geometric Relations Table, which is a utility under the REVgraph. The PGRT delivers pairs of lines to the user satisfying certain user defined geometric criteria. For each such pair a number of geometric relations are calculated, which are cached in a table should that pair be requested again.

At this stage, the user may also define pairs of wires to be placed on the interesting-pairs list, but this is not necessary for the algorithm to run.

The Algorithm

The algorithm is object oriented, and arranged in a pass structure. In each pass, each of the wires in the interesting-wires list attempts to find a completion for both of its ends, in a vertex or T junction or collinearity. Between each pass, an optimal context is found, which is the basis for further completions. Also between each pass it is possible for the top down directives to be modified as the user desires.

The method we used to decide on the optimal context in this problem is very straightforward. The aim of the algorithm is to connect wires together, so the most successful context is the one in which most wire ends are connected to something. Whenever a contradiction is found, the context with the most objects is chosen for further exploration. Such contexts are often suboptimal, but this strategy gives the search a necessary breadth during each pass of the algorithm.

For each end of each interesting wire the following criteria are used to choose a new junction to create :

If the wire end is already linked to something in the current context, then no attempt is made to create a new junction. Otherwise, the candidate list of the wire end is examined as follows.

First, all those candidates which have already been used to create a junction are ignored. If there is a remaining candidate pair which is tagged as having been connected before segmentation, then that is chosen. Otherwise, any pair which appears to be parallel, close and overlapping is chosen. If there is still no successful candidate, then the interesting-pairs list is searched for the presence of any remaining candidates, and the first such candidate found will be chosen. Finally, if all else fails, the pair which would produce the junction closest to the wire end is chosen.

Junction types and constraints

When a pair is selected to be instantiated as a junction, the geometric relations provided by the PGRT are used to determine junction type. Basically, measures of collinearity, and distance between wires at their image crossing point are used to place the junction in one of these categories : Collinearity, Bar, Vertex, T junction, and Maybe-Vertex.

For a collinearity, a single connector is created, each end of which is attached to the relevant end of the two wires. The new connector is heuristically justified by the conjunction of the two wires.

For a bar, where two lines are close, parallel and overlapping, then no connector is created, but the two lines are linked directly to one another. Effectively, this provides a rigorous justification for the link, as both wires are rigorous axioms, but the link itself is not explicitly represented as an object.

For a vertex, two new connectors and a new vertex are created. The vertex is positioned in x and y according to where the wires cross on the image plane, and in depth half way between the two wires' depths at this point. The vertex is heuristically justified by the conjunction of the two wires. Each connector is rigorously justified by the vertex. If the vertex is found (by searching) to be sufficiently close to another vertex, then the new vertex is incorporated into the relevant super-vertex. The super-vertex is then heuristically justified by the new vertex.

As often happens with trihedral vertices, one wire will end up being connected to two vertices which are judged to be in the same super-vertex. Thus each vertex has one wire in common. In this case, a new connector is not created for that wire, but the existing one is re-used, and given a new justification.

For T junctions one new connector is created, and one T junction object. The connector goes between the occluded wire and the T junction. The new connector is rigorously justified by the T junction. The occluded wire may well not be occluded by the other wire directly, but rather by a connector linking that other wire to another junction of some kind. Several such connectors may exist, in different contexts, so the T junction requires a new justification for each such occluding connector that is created.

Every time a new connector is created, it is tested against all the relevant T junctions to determine if it occludes any of them. If so, the T junction is given an extra heuristic justification of the conjunction of the occluded wire and the connector.

Similarly, whenever a new T junction is created then it is tested against all the relevant connectors, and appropriate justifications are made.

If the junction type is Maybe-Vertex, then initially a T junction is created (if possible), but if a vertex is ever created close enough to this T junction, then this is taken as enough evidence for the T junction actually being a vertex, and a new vertex is created to replace the original T junction. The new vertex is then considered to be part of the same super-vertex as the vertex which was justified the upgrading of the T junction.

If a junction type is a vertex, but the extension of one wire meets the other along its length, then it is topologically convenient to represent this as a T junction. However, if such a T junction is close enough to the end of the other wire, then a vertex is created with one connector going back along the length of the wire.

The sequence of events on creating a new junction is as follows. First, all the new objects required for the junction are created, added to the lists of the relevant object type, set to point to objects they are linked to, and geometrically initialised. This thoroughly imbeds new objects into the PWFC data structure. Then, the objects must be introduced to the CMS, which involves a justification request for each new object. At this stage, any new connectors are checked against all T junctions, and any new T junctions are checked against all connectors, to see if any new justifications can be found for the relevant T junctions. The final phase is to check for contradictions, as described below. When all this has been done, a new wire end is chosen, and the process of completion starts again.

Contradiction Constraints

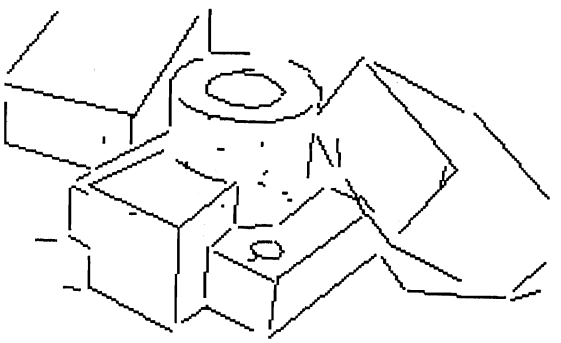
Any two connectors which occupy the same end of the same wire are made contradictory to each other. Any connector pair or wire and connector pair which cross on the image plane are marked as contradictory, with an exception in special circumstances. These are the two "rules" which deal with contradictions, and they are applied whenever a new connector is created.

These two rules are applied at very specific points in the program. Whenever a new connector is created, any wire it is linked to has its end checked for other links. Any other object linked to the relevant end of the wire is made contradictory to the newly created connector.

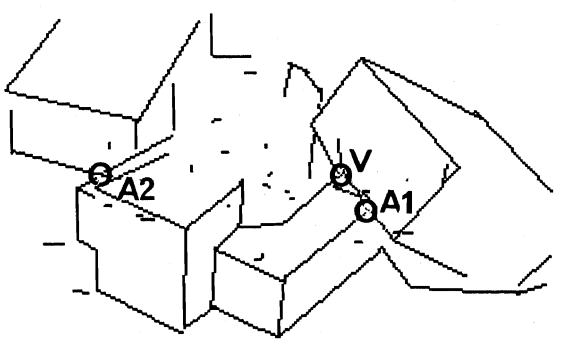
Whenever a new connector is created, it is tested against all other wires and connectors in the database to see if it crosses any of them. Every other wire and connector it crosses is made contradictory to the new connector. Such an exhaustive search for crossing objects is not

totally necessary, and a little more time consuming than is desirable.

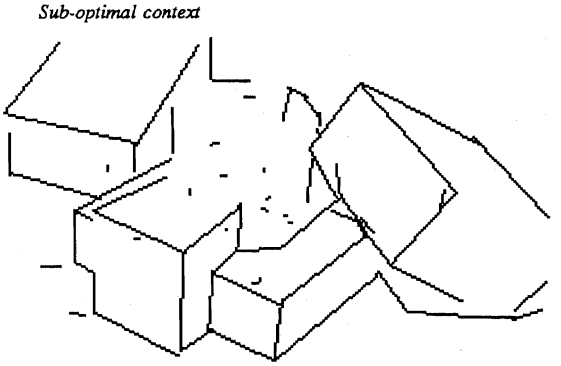
The exception to the second "rule" is when the two objects are connected to vertices which are part of the same super-vertex. The geometrical description of lines is such that the centroid of the line is known to a certain error, and the direction of the line is known to a certain error. This has the effect that the greatest positional errors are at the line ends. Consequently, when connecting up vertices of 3 or more wires, which are perfectly valid, it is often found that the component connectors in the individual 2 way vertices cross over each other, and cross over component wires. Thus the necessary exception to the connector crossing rule.



Original Data



Sub-optimal context



Optimal Context

Figure 5 :- Before and After Wireframe Completion

Results

Figure 5 shows the initial data from one image for Wireframe Completion, and 2 contexts after four passes of the algorithm. During the fifth pass, no more junctions were found, so pending further top down information processing could be said to be complete after the fourth pass. The bottom context is adjudged to be optimal, while the middle one highlights the major ambiguities.

The image contains several lines with no depth information, which account for most of the missed vertices, and some T junctions which should be vertices. The interesting wires were set to be all the wires with some depth information. The limiting radius was set to 15 image pixels, which is relatively large and likely to make the algorithm thrash a little more than is necessary. The circular arcs which should be present round the top of the central object are missing in this straight-line-only version of the algorithm. Their inclusion in the algorithm should cause several more vertices to be found.

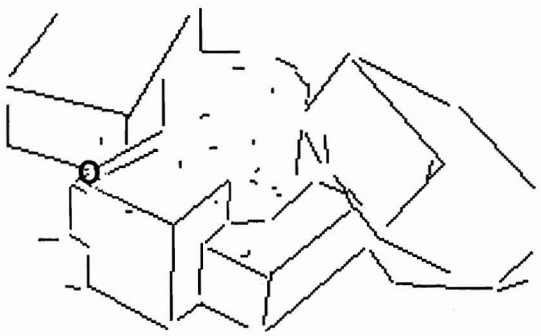
Most of the work was done during the first pass of the algorithm, which confirms our prediction that most of the junctions are obvious, with only a few ambiguities. In total, three ambiguities were found : On the far right hand corner of the central object three T junctions are found to be alternative solutions to the local connections, though one in particular is considered optimal (labelled A1 on Figure 5). On the left hand side of the same object, three alternative solutions were eventually found, with the intuitively best solution being picked as optimal by the program. (labelled A2 on Figure 5).

The development of this latter ambiguity as the program runs is interesting, and is shown in detail in Figure 6. Initially, the worst solution is found, where the straight line on the left hand object is extended through a break in the bounding line of the central object to form a spurious T junction. (Figure 6b). This is found to be inadequate, as the bounding contour cannot be completed in such a context, so a second solution is found, which involves a T junction and vertex in close proximity, and one line segment being completely unconnected. (Figure 6c). The unconnected line segment runs close and parallel to the new connector, so close that in the display they run into each other, appearing in the figure as one thicker line rather than two distinct lines. Finally, the "correct" solution is found. (Figure 6d).

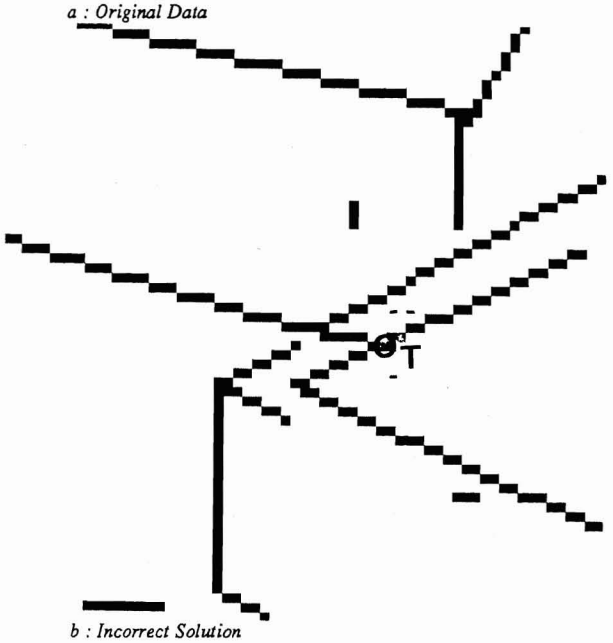
There are also some unfortunate errors in this sequence of processing. One particularly unpleasant feature is the vertex on the right hand block which involves an extension of one of the wires from the central object (labelled V on Figure 5). That wire should be occluded by the short bounding line at the rear of that object, but unfortunately there was insufficient depth information to make any completions to that line, so the spurious vertex remained.

The total number of contexts produced as a result of these ambiguities was six. The time spent by the CMS processing the various constraints was less than, though of the same order as the time spent searching for junctions and

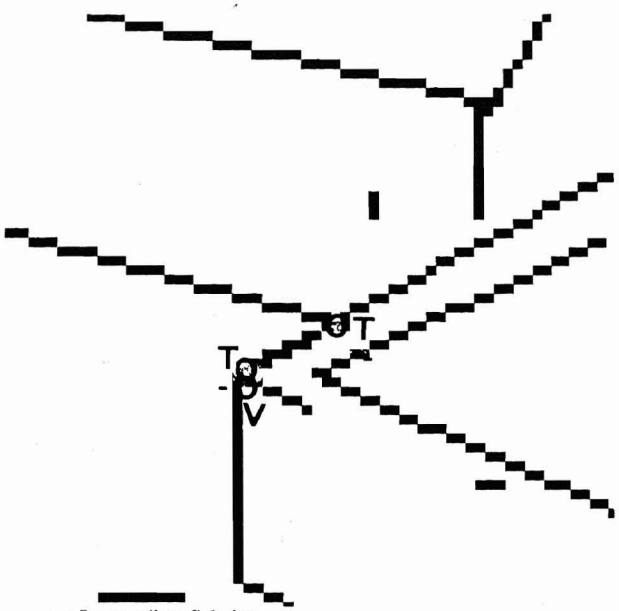
creating the necessary data structures. While we will admit that very careful thought had to be given to the structuring of the constraints, and that this particular example problem might have been more efficiently processed using a conventional breadth and bound search technique, we feel we have shown the potential value of a CMS in the vision domain, where ambiguity is comparatively infrequent, and caused by failings in heuristics rather than contradictory data. Higher level reasoning schemes which are more processor intensive, such as might attempt to start assigning surfaces to completed regions in the example would possibly benefit greatly by utilising a TMS such as this.



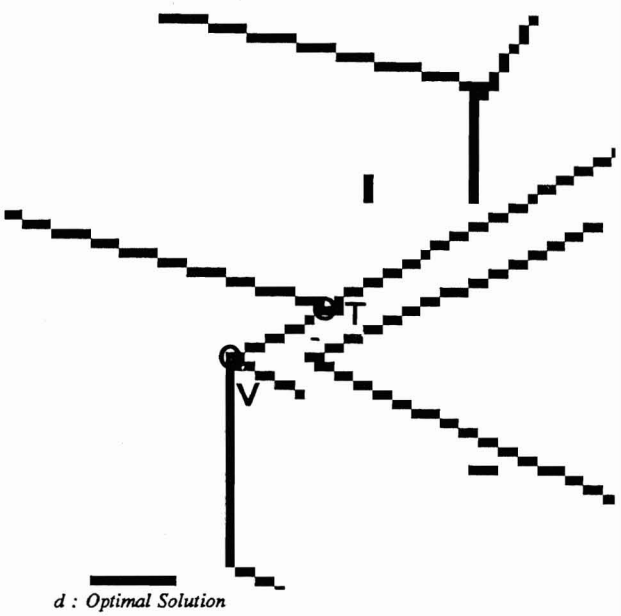
a : Original Data



b : Incorrect Solution



c : Intermediate Solution



d : Optimal Solution

Figure 6 : 3 Solutions to Ambiguity A2 in Figure 5

References :

1 Porrill JP, SB Pollard, TP Pridmore, JB Bowen, JEW Mayhew, JP Frisby (1987) TINA : The Sheffield AIVRU vision system, AIVRU memo 027.

2 Pollard SB, JEW Mayhew and JP Frisby (1985) PMF: A stereo correspondence algorithm using a disparity gradient limit, Perception 14, 449-470.

3 Pridmore TP, J Porrill, JEW Mayhew and JP Frisby (1985) Geometrical description of the CONNECT graph #1 : Straight lines, planes, space curves and blobs, AIVRU memo 011.

4 Doyle J (1979) A Truth Maintenance System , Artificial Intelligence 12 , 231-272.

5 De Kleer J (1984) Choices without backtracking , Proceedings National Conference on Artificial Intelligence, August 1984.

6 McDermott D (1983) Contexts and Data Dependencies : A synthesis , IEEE Pattern analysis and machine intelligence.

7 De Kleer J (1986) An Assumption-based TMS , Artificial Intelligence 28 , 127-161.