MIT/LCS/TR-76

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick H. Winston

September 1970

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick H. Winston

September 1970

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts 02139

ACKNOWLEDGMENT

Marvin Minsky, Janet Winston, and Seymour Papert contributed significantly to the technical content of this work.

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(02).

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES*

Abstract

The research here described centers on how a machine can recognize concepts and learn concepts to be recognized. Explanations are found in computer programs that build and manipulate abstract descriptions of scenes such as those children construct from toy blocks. One program uses sample scenes to create models of simple configurations like the three-brick arch. Another uses the resulting models in making identifications. Throughout emphasis is given to the importance of using good descriptions when exploring how machines can come to perceive and understand the visual environment.

---

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degree of Doctor of Philosophy, January 1970.

Table of Contents

1 Key Ideas

How do we recognize examples of various concepts?

How do we learn to make such recognitions?

How can machines do these things?

How important is careful teaching?

In this paper I describe a system that sheds some light on these questions by demonstrating how a machine can be taught to see and learn new visual concepts. It works in the domain of three-dimensional structures made of bricks, wedges, and other simple objects.

Good descriptive methods are of central importance to this work. This is demonstrated repeatedly in my system's facilities for scene description, description comparison, concept learning, and identification.

It is my opinion that the framework for learning that I describe suggests a unity between learning from examples, learning by imitation, and learning by being told. This unity lies in the necessary ability to generate and manipulate good abstract descriptions.

I also argue the importance of good training sequences prepared by good teachers. I think it is reasonable to believe that neither machines nor children can be expected to learn much without them.

Equally important is the notion of the near miss. By

near miss I mean a sample in a training sequence quite like the concept to be learned but which differs from that concept in only a few significant points at most. These near misses prove to convey essential points much more directly than repetitive exposure to ordinary examples.

1.1 Scene Description and Comparison

Much of the system to be described focuses on the problem of analyzing scenes consisting of the simple objects that one finds in a child's toy box. There are two very simple examples of such scenes in figure 1-1.

From such visual images, the system builds a very coarse description. (figure 1-2) Structuring the scene's description in terms of objects is already a certain commitment, for structuring it in other terms is possible. In any case, analysis proceeds, inserting more detail. (figure 1-3) And finally there is the very fine detail about the surfaces, lines, vertexes, and their relations.

Such descriptions permit one to compare and contrast scenes through programs that compare and contrast descriptions. Of course, one hopes that the descriptions will be similar or dissimilar to the same degree that the scenes they represent seem similar or dissimilar to human intuition. Then with a general plan for such manipulations, there is further hope that the same machinery can be useful

FIGURE 1-1

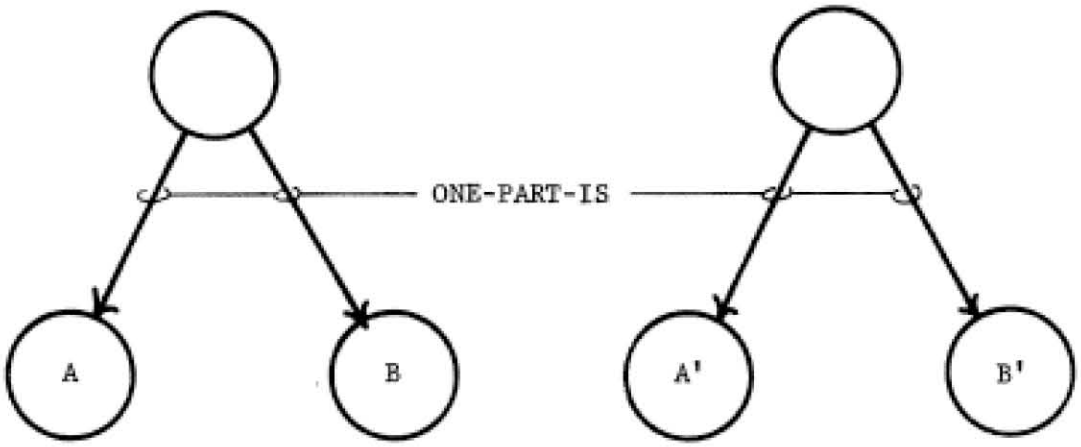


FIGURE 1-2

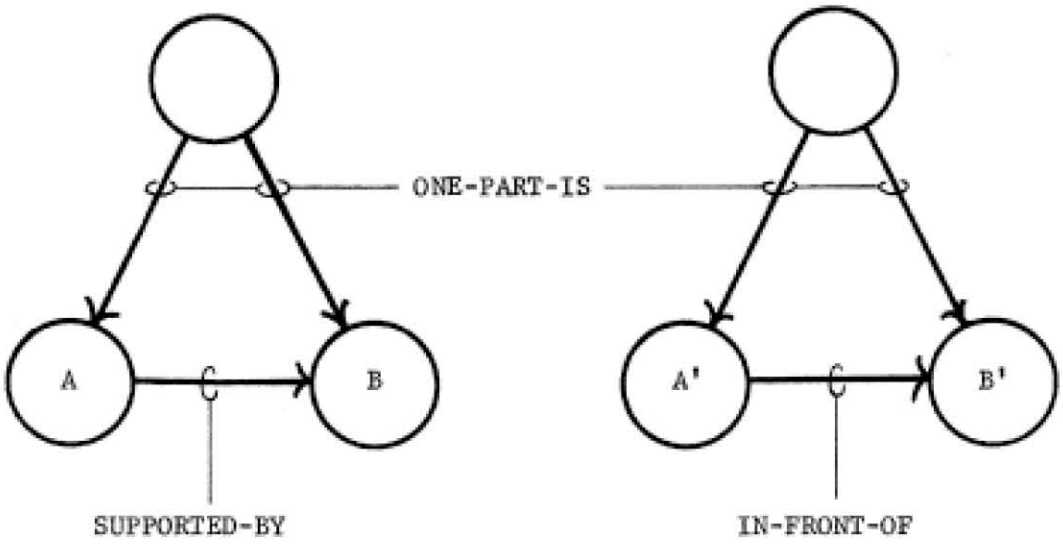


FIGURE 1-3

in situations ranging far from visual ones, giving the work a certain generality.

Certainly the necessary matching programs must be well endowed with ability, for a rich description capability requires a matching program that can cope with and perform reasonably in an environment where many matches are possible, both good and bad.

After two scenes are described and corresponding parts related by the matching program, differences in the descriptions must be found, categorized, and themselves described. The program that does this must be able to examine the descriptions of figure 1-3 with the help of a matching program and deduce that the difference between the scenes is that there is a supported-by relation in one case, while there is an in-front-of relation in the other. But the faculty must be much more powerful than this simple example indicates in order to face more complex pairs of scenes exhibiting the entire spectrum between the nearly identical and the completely different.

1.2 Concept Generation and Learning

To build a machine that can analyze line drawings and build descriptions relevant to some comparison procedure is useful in itself. But this is just a step toward the more ambitious goal of creating a program that can learn to

ARCH

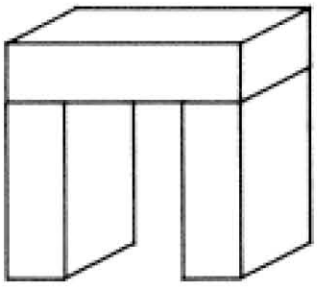


FIGURE 1-4

NEAR MISS

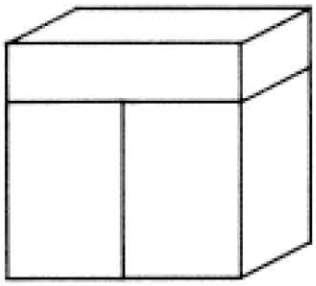


FIGURE 1-5

ARCH

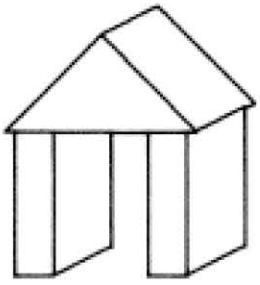


FIGURE 1-6

NEAR MISS

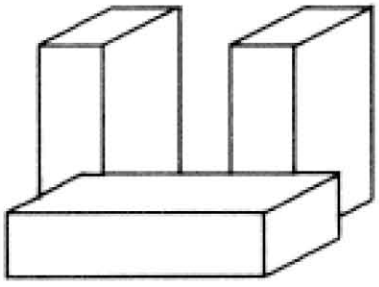


FIGURE 1-7

recognize structures. I will describe a program that can use samples of simple concepts to generate models.

Figure 1-4 and the next few following it show a sequence of samples that enables the machine to learn what an arch is. First it gets the general idea by studying the first sample in figure 1-4. Then it learns refinements to its original conception by comparing its current impression of what an arch is with successive samples. It learns that the supports of an arch cannot touch from figure 1-5. It learns that it does not matter much what the top object is from figure 1-6. And then from figure 1-7 it learns the fact that for one object to be supported by the others is a definite requirement, not just a coincidence carrying through all of the samples.

Such new concepts can in turn help in making other, more complex abstractions. Thus the machine uses previous learning as an aid toward further learning and further analysis of the environment. As yet these procedures are clumsy, and the descriptions uncomfortably restricted, but the results are encouraging enough to suggest that these methods may lead to increasingly powerful performance.

1.3 Identification

Identification requires additional programs that use the results of comparison programs. There are many problems and many alternative methods involved because identification can be done in a variety of contexts.

In one simple form of identification, the machine compares the description of some scene to be identified with a repertoire of models, or stored concepts. Then at the very least there must be some method of evaluating the comparisons between the unknown and the models so that some match can be defined as best.

But many sophistications lie beyond this skeletal scheme. For one thing, the identification can be sensitive to context. In figure 1-8, for example, one hidden object is more likely to be a wedge than in the other case, although both hidden objects present exactly the same line configuration. The identification could be further prejudiced if the objective is to locate a particular type of object. Thus the hidden object in figure 1-9 should be tentatively identified as a possible trapezoidal solid, rather than a wedge, if trapezoidal solids are in demand.

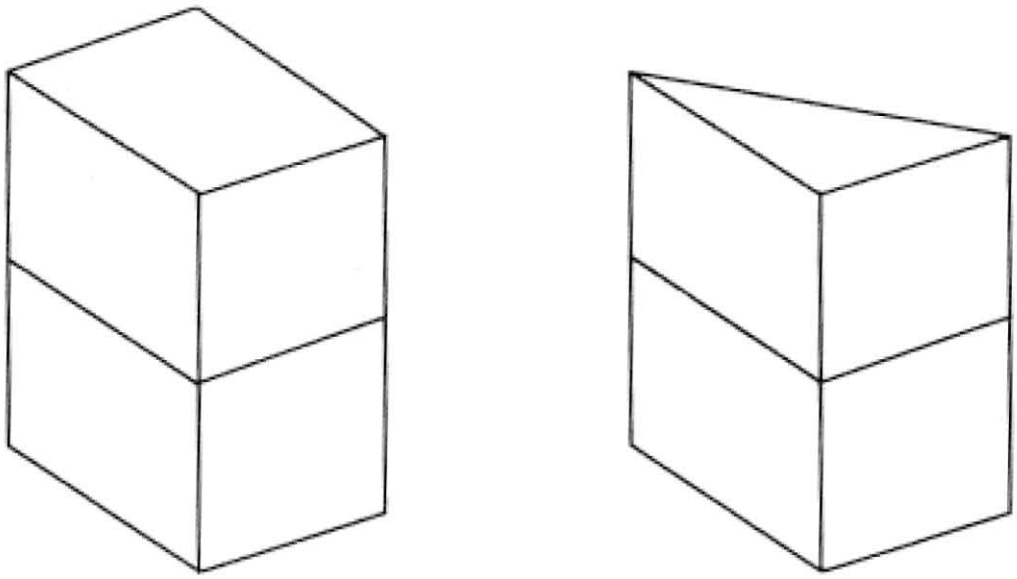


FIGURE 1-8

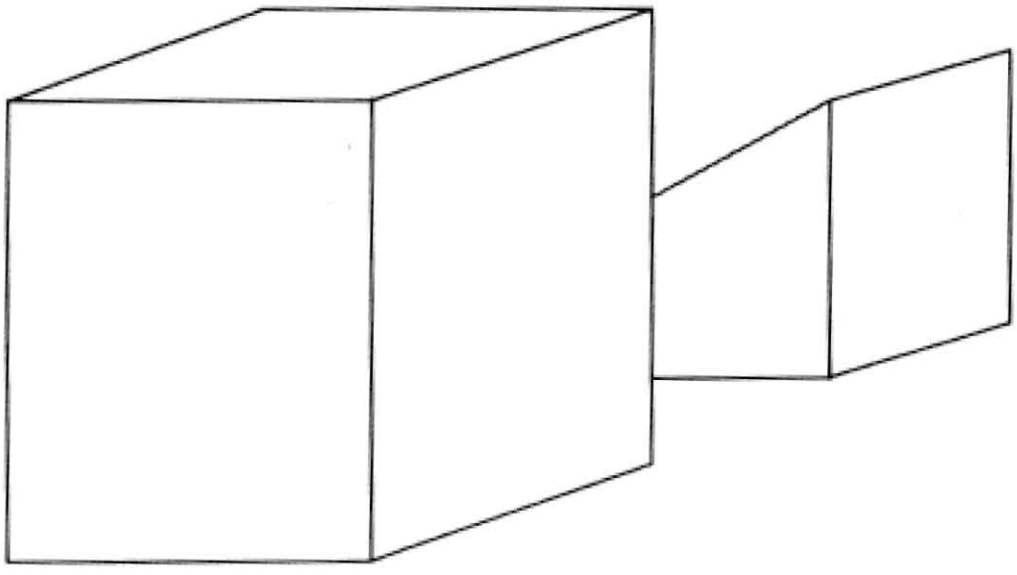


FIGURE 1-9

1.4 Psychological Modeling

Simulation of human intelligence is not a primary goal of this work. Yet for the most part I have designed programs that see the world in terms conforming to human usage and taste. These programs produce descriptions that use notions such as left-of, on-top-of, behind, big, and part-of.

There are several reasons for this. One is that if a machine is to learn from a human teacher, then it is reasonable that the machine shoud understand and use the same relations that the human does. Otherwise there would be the sort of difference in point of view that prevents inexperienced adult teachers from interacting smoothly with small children.

Moreover, if the machine is to understand its environment for any reason, then understanding it in the same terms humans do helps us humans to understand and improve the machine's operation. Little is known about how human intelligence works, but it would be foolish to ignore conjectures about human methods and abilities if those things can help machines. Much has already been learned from programs that use what seem like human methods. There are already programs that prove mathematical theorems, play good chess, work analogy problems, understand restricted forms of English, and more. Yet in contrast, little knowledge about

intelligence has come from perceptron work and other approaches to intelligence that do not exploit the planning and hierarchical organization that is characteristic of human thought.

Another reason for designing programs that describe scenes in human terms is that human judgement then serves as a standard. There will be no contentment with machines that only do as well as humans. But until machines become better than humans at seeing, doing as well is a reasonable goal, and comparing the performance of the machine with that of the human is a convenient way to measure success.

2 Building Descriptions

2.1 The Network

There are many ways to store facts about a scene. One simple format is the unordered list:

A is on top of B

A1 is a side of A

B is in front of C

Such an arrangement is desperately inefficient because the whole of memory must be searched to gather all facts about some particular component of the scene. It is natural, therefore, to record facts in a more structured way to facilitate retrieval.

In this connection, one hears such terms as lists, trees, rings, and nets, each of which suggests a form of storage. In selecting one, attention must be paid to several criteria. I have already mentioned the problem of rapid access. There may also be a need to use memory space efficiently. But in the research phase, perhaps it is most important that the storage format be in some sense natural with respect to the information to be stored. This means that the transformation from a situation to its representation should be simple, not awkward. Simple lists suffice for a trip to the grocery store, while tree-like charts frequently picture command hierarchies or genealogical

histories.

But many more complex situations require the net. A good example is the description of the words in a natural language. Each word is described easily in terms of relationships with other words which in turn are similarly described. The result is a dictionary in which each word may be thought of as a node which is related to other nodes through the pointers that constitute its definition.

Similarly the network seems to have the appropriate blend of flexibility and elegance needed to deal straightforwarly with scenes. It is the natural format. Like words in a dictionary, each object is naturally thought of in terms of relationships to other objects and to descriptive concepts like large, rectangular, and standing. In figure 2-1, for example, one has concepts such as OBJECT-ABC and OBJECT-DEF. These are represented diagrammatically as circles. (figure 2-2) Labelled arrows or pointers define the relationships between the concepts. (figure 2-3) Other pointers indicate membership in general classes or specify particular properties. (figure 2-4) And pointers to circles representing the sides extend the depth of the description and allow more detail. (figure 2-5)

Now notice that notions like SUPPORTED-BY, ABOVE, LEFT-OF, BENEATH, and A-KIND-OF may be used not only as relations,

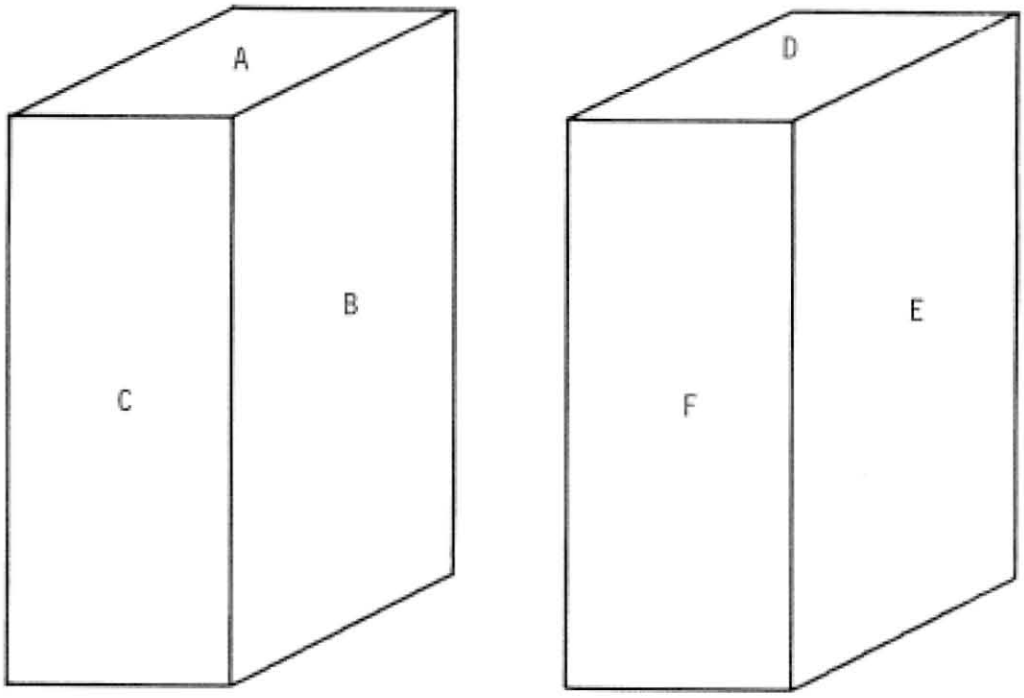


FIGURE 2-1

FIGURE 2-2

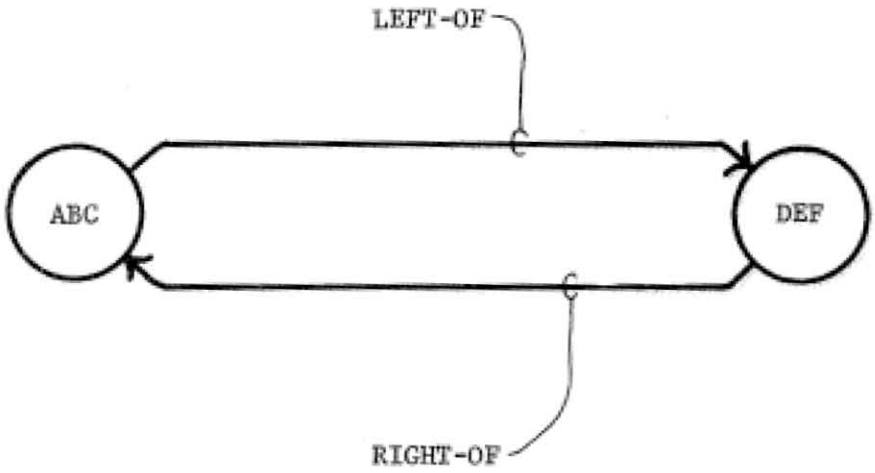


FIGURE 2-3

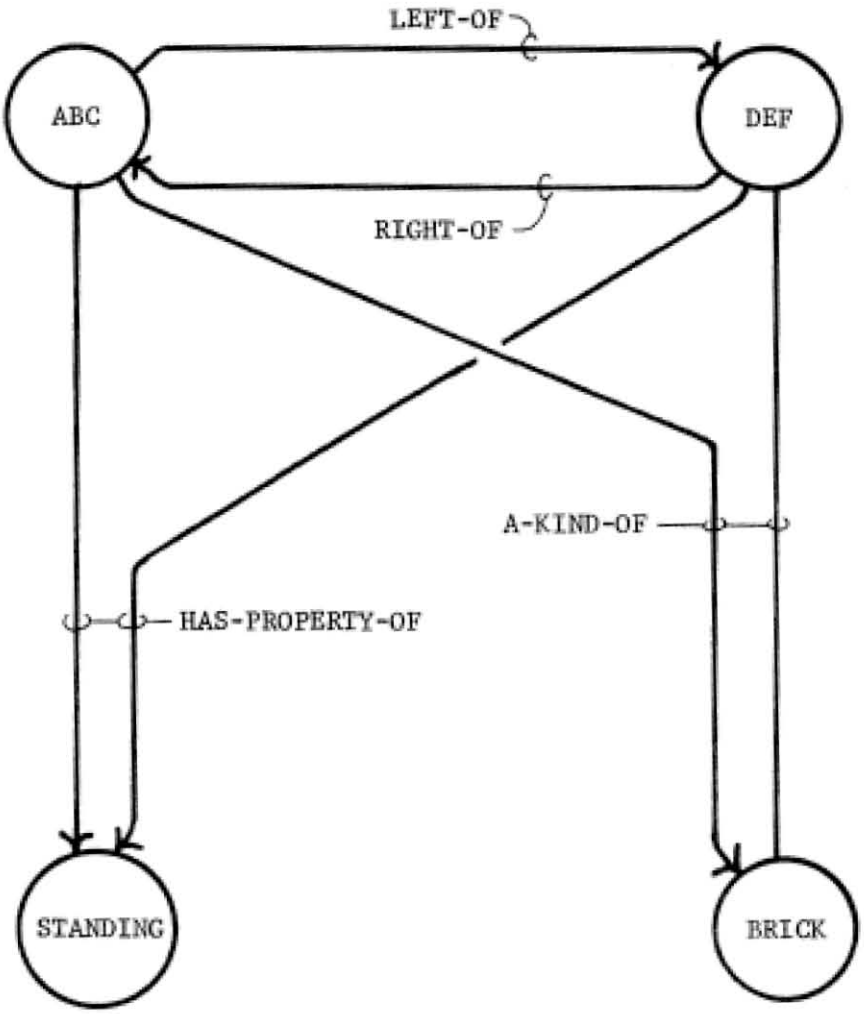

LEFT-OF
ABC
DEF
RIGHT-OF
A-KIND-OF
HAS-PROPERTY-OF
STANDING
BRICK

FIGURE 2-4

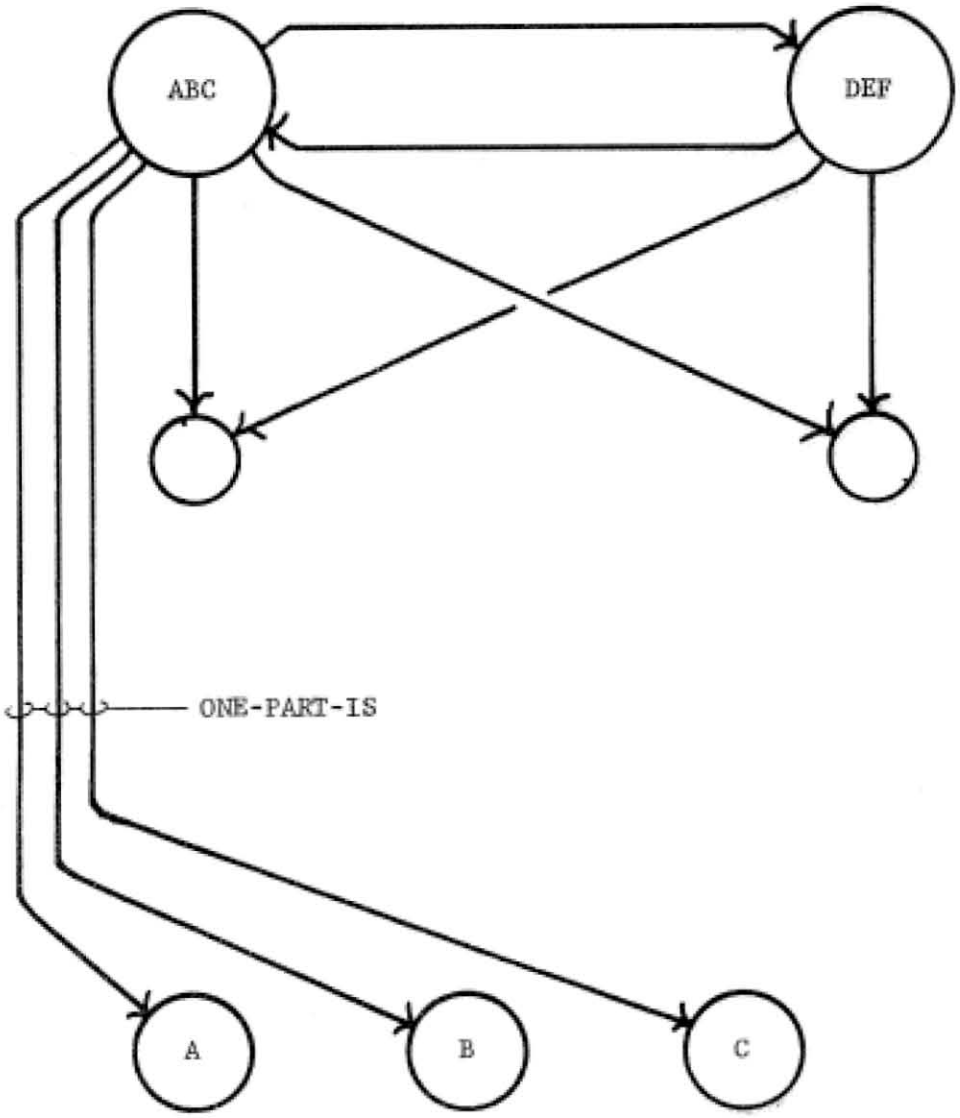


FIGURE 2-5

but also as concepts. Consider SUPPORTED-BY. The statement, "The WEDGE is SUPPORTED-BY the BLOCK," uses SUPPORTED-BY as a relation. But the statement, "SUPPORTED-BY is the opposite of NOT-SUPPORTED-BY," uses SUPPORTED-BY as a concept undergoing explication. Consequently SUPPORTED-BY is a node in the network as well as a pointer label, and SUPPORTED-BY itself is defined in terms of relations to other nodes. Figure 2-6 shows some of the surrounding relations and concepts.

SUPPORTED-BY may therefore appear in diagrams as a circle label or as a pointer label depending on its function. A circle pierced by an arrow indicates simultaneous use as a relation and as a concept. (figure 2-7)

Thus, descriptions of relationships can be stored in a homogeneous network along with the descriptions of scenes that use those relationships. This permits big steps toward program generality. A program to find negatives need only know about the relation NEGATIVE-SATELLITE and have access to the general memory net. There is no need for the program itself to contain a distended table. This way programs can operate in many environments, both anticipated and not anticipated. Algorithms designed to manipulate networks at the level of scene description can work as easily with descriptions of objects, sides, or even of objects'

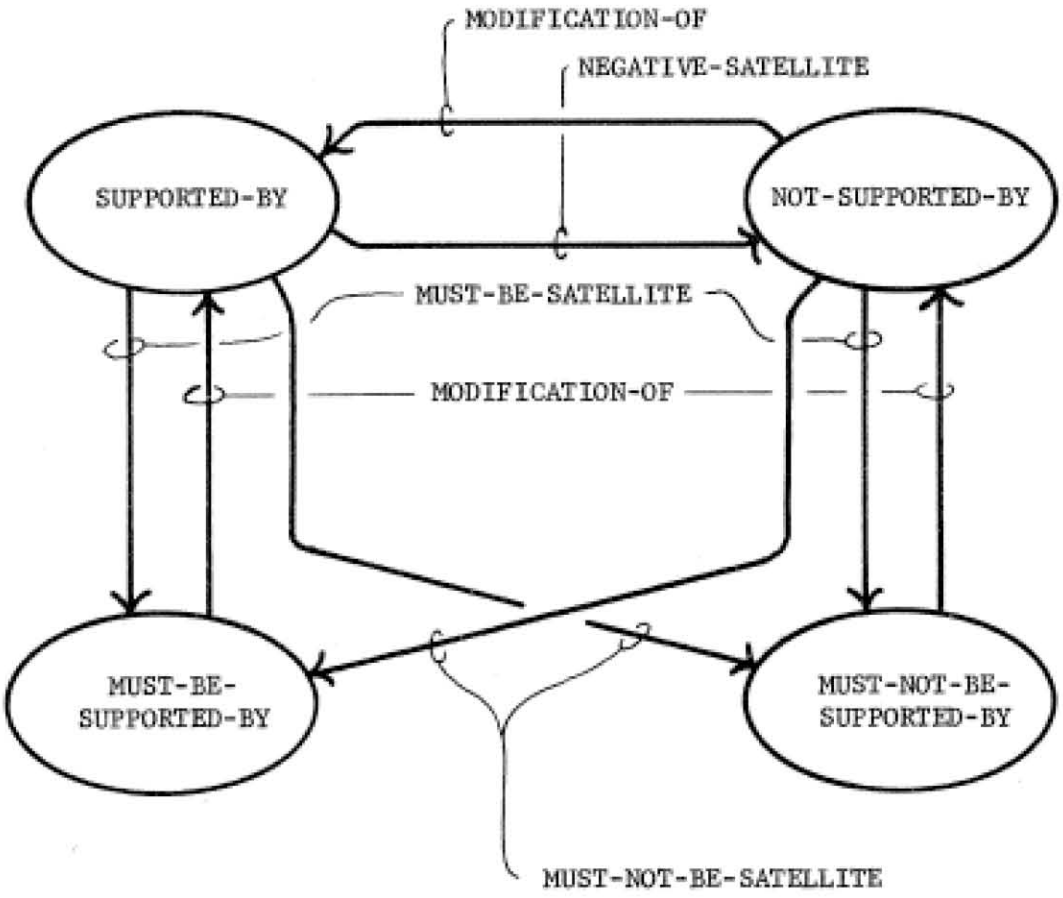

MODIFICATION-OF
NEGATIVE-SATELLITE
SUPPORTED-BY
NOT-SUPPORTED-BY
MUST-BE-SATELLITE
MODIFICATION-OF
MUST-BE-
SUPPORTED-BY
MUST-NOT-BE-
SUPPORTED-BY
MUST-NOT-BE-SATELLITE

FIGURE 2-6

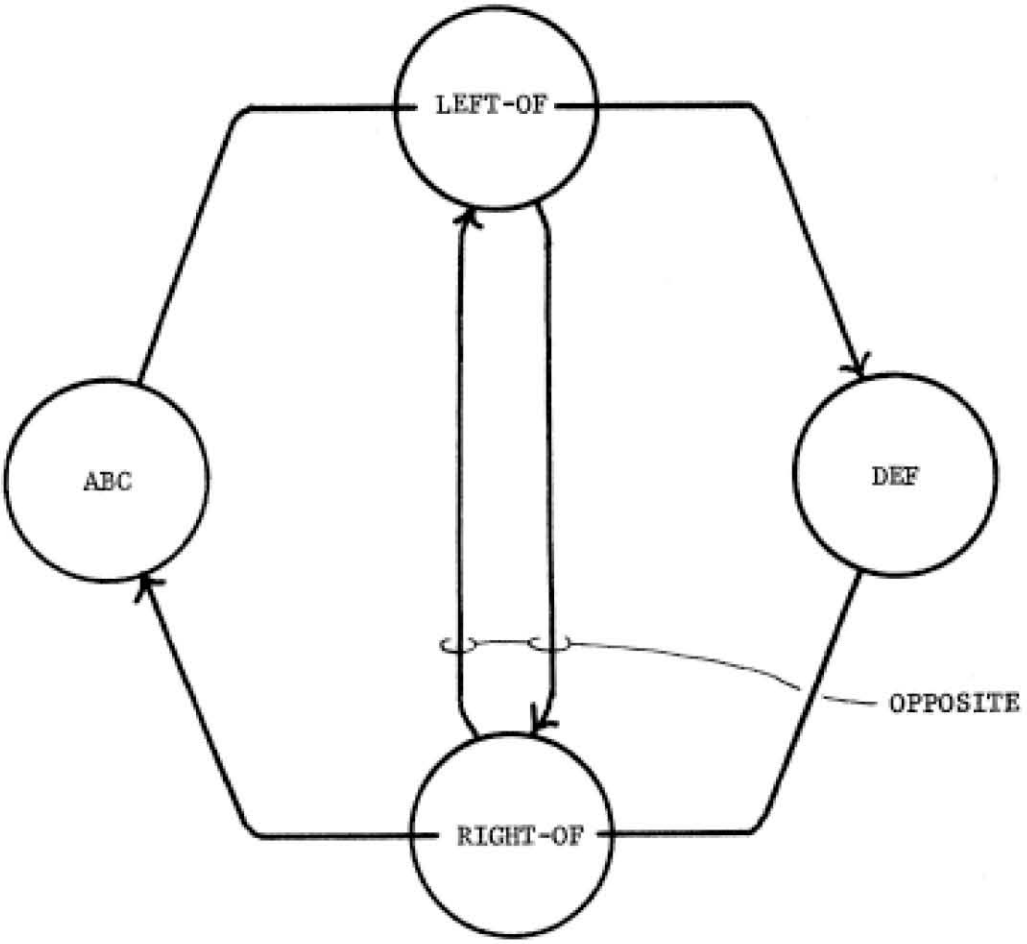


FIGURE 2-7

functions, given the appropriate network.

2.2 Preliminary Processing

Consider now the generation of a scene description. The starting point is a line drawing, without perspective distortion, and the result is to be a network relating and describing the various objects with pointers such as IN-FRONT-OF, ABOVE, SUPPORTED-BY, A-KIND-OF, ABUTS, and HAS-PROPERTY-OF.

First, drawings of three-dimensional scenes are communicated to the machine using a program by B. K. P. Horn together with a special pen whose position on a companion tablet can be read by the machine directly. Then a program written by H. N. Mahabala [1] classifies and labels the vertexes according to the number of converging lines and the angles between them. Figure 2-8 displays the available categories. Notice that Mahabala's program finds pairs of Ts where the crossbars lie between collinear uprights. These are called matched Ts. Such pairs occur frequently when one object partially occludes another as in figure 2-9.

The program then proceeds to create names for all of the regions in the scene. Rigorously "region" as used here simply refers to any maximal area in which one can move from any point to any other point without crossing a line. Including the background figure 2-9 has eight regions. Various

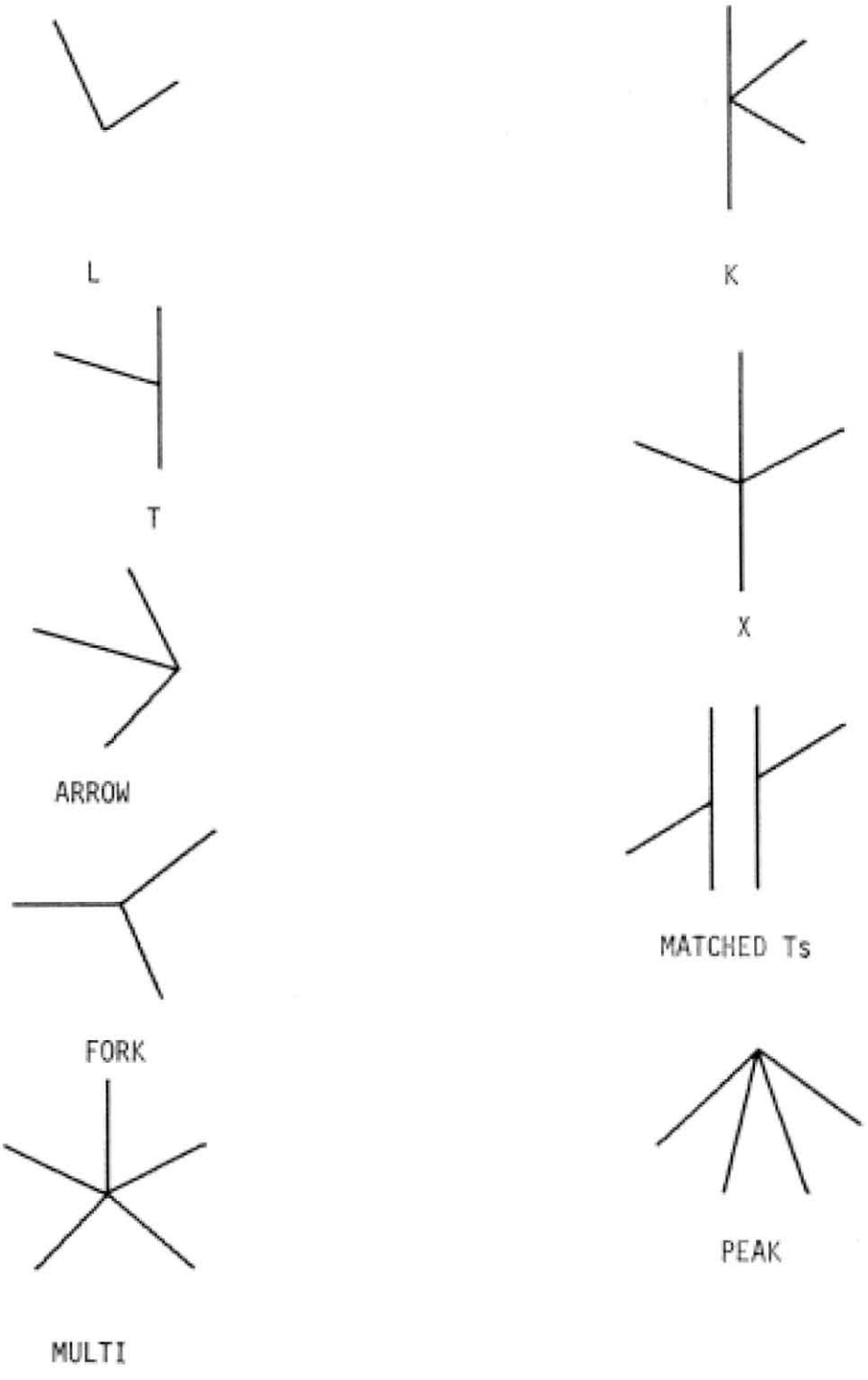


FIGURE 2-8

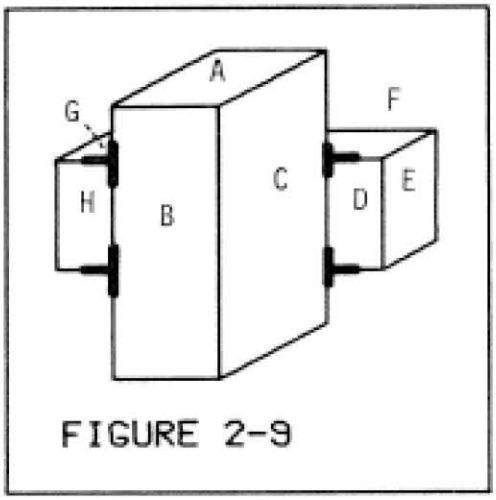


FIGURE 2-9

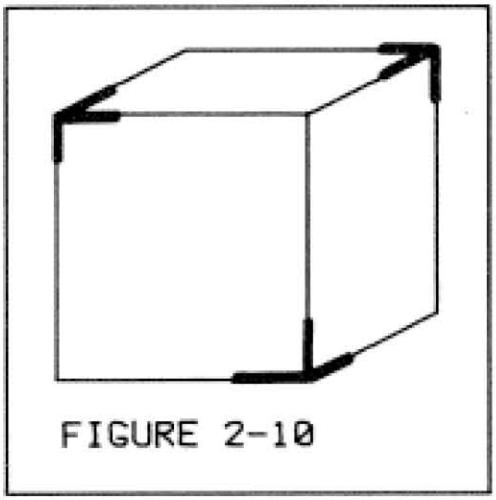


FIGURE 2-10

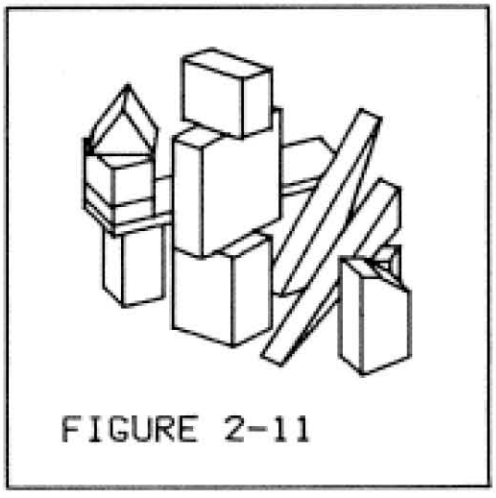


FIGURE 2-11

properties are calculated and stored for these regions. Among these are a list of the vertexes surrounding each region and a list of the neighboring regions.

These results are then supplied to the elegant program named SEE developed by A. Guzman [2]. This program conjectures about which regions belong to the same objects. For figure 2-9, the end result of the program is the commentary:

Body 1 consists of A B C

Body 2 consists of D E F G H

Surprisingly the program contains no explicit models for the objects it expects to see. It simply examines the vertexes and uses the vertex classifications to determine which of the neighboring regions are likely to be part of the same object. Arrows, for example, strongly suggest that the two narrow-angle regions belong to the same body. (figure 2-10) This sort of evidence, together with a moderately sophisticated executive, can sort out the regions in scenes as complicated as that in figure 2-11, borrowed from Guzman's thesis. Twelve objects are reported and the regions of each are remembered.

This, then, is the sort of information ready for further processing by my description-building programs.

2.3 The Algorithms

The following sections describe the ideas behind programs that look for the relations ABOVE, SUPPORT, IN-FRONT-OF, LEFT, RIGHT, and MARRYS. Generally these programs produce descriptions that are in remarkable harmony with those of human observers. Sometimes, however, they make conjectures that most humans disagree with. On these occasions one should remember that there is no intention to precisely mimic psychological phenomena. The goal is simply to produce reasonable descriptions that are easy to work with. Right now it is important to design and experiment with a capable set of programs and postpone the question of how the programs might be refined to be more completely lifelike.

2.3.1 Above and Support

T joints are strong clues that one object partly obscures another, but then one may ask if the obscuring occurs because one object is above the other or because one is in front of the other. Even in the simple two brick case there seems to be an enormous number of configurations. Figure 2-12 shows just a few possibilities.

But in spite of this variety, there is a simple procedure that often seems to correctly decide the ABOVE versus IN-FRONT-OF question. Consider the lines that form

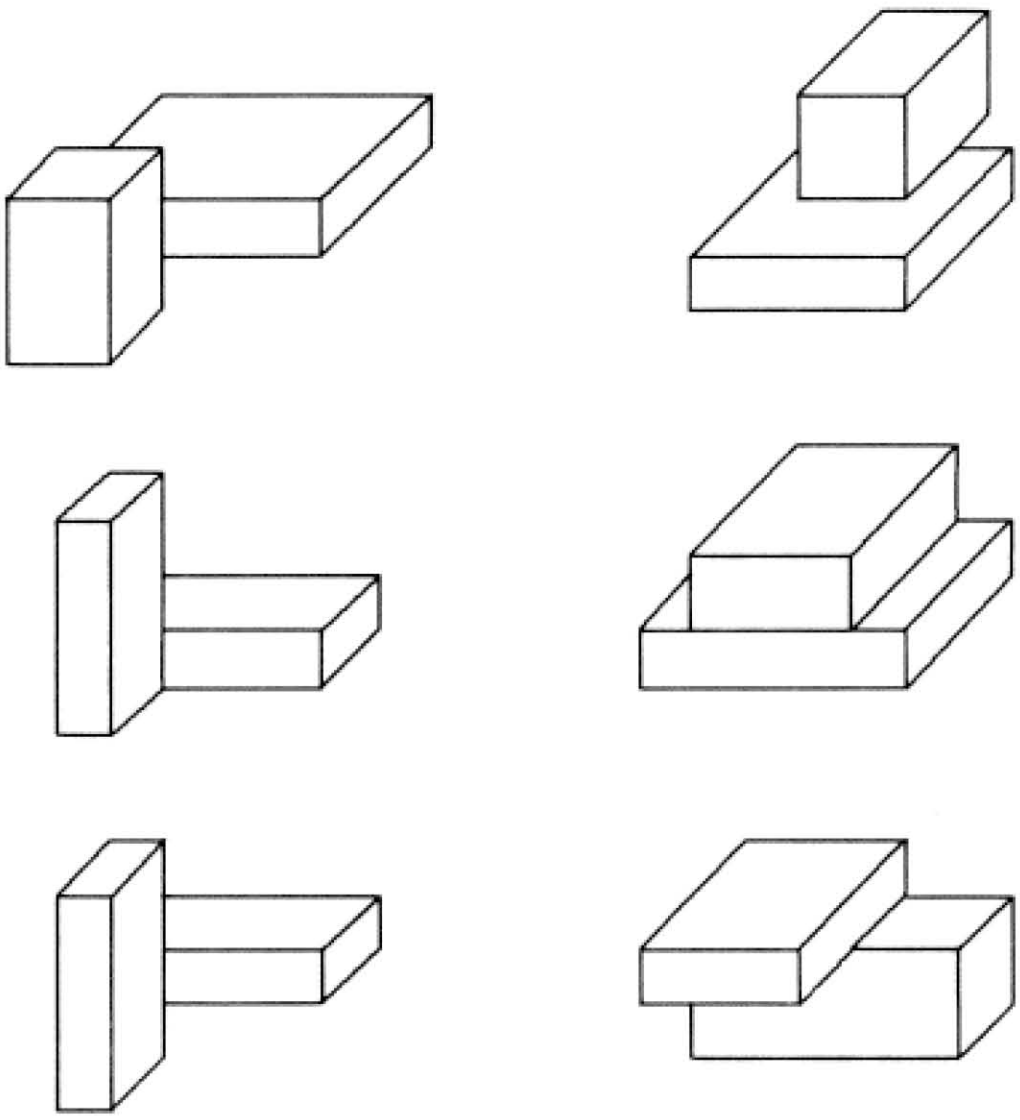


FIGURE 2-12

the bottom border of the obscuring objects in figure 2-12. Finding these lines is the first job of the program. Next the program finds other objects whose regions share these lines. In general these other objects are below the original, obscuring object.

This algorithm works on all the simple two-block situations depicted in figure 2-12. It even works correctly on the much more complicated, many-object scene in figure 2-13, shown with the bottom lines highlighted.

The difficult part is to find the so-called bottom lines, which correspond roughly to one's intuitive notion of bottom border. The process proceeds by first noting those lines that lie between two regions of the object in question. I call these interior lines. Next the program examines the lower of each interior line's vertexes. This is ignored unless it is an arrow, X or a K. Then information about bottom lines is gleaned from each of the arrows, Xs, and Ks in the following way:

1. If the vertex is an arrow, then the two lines forming the largest angle, the barbs, are bottom line candidates. See figure 2-14.

2. If the vertex is an X, then the two non-collinear lines are bottom line candidates. See figure 2-15.

3. If the vertex is a K, then the two adjacent lines, those forming the smallest clockwise and the smallest counter-clockwise angles with the interior line are bottom line candidates. See figure 2-16.

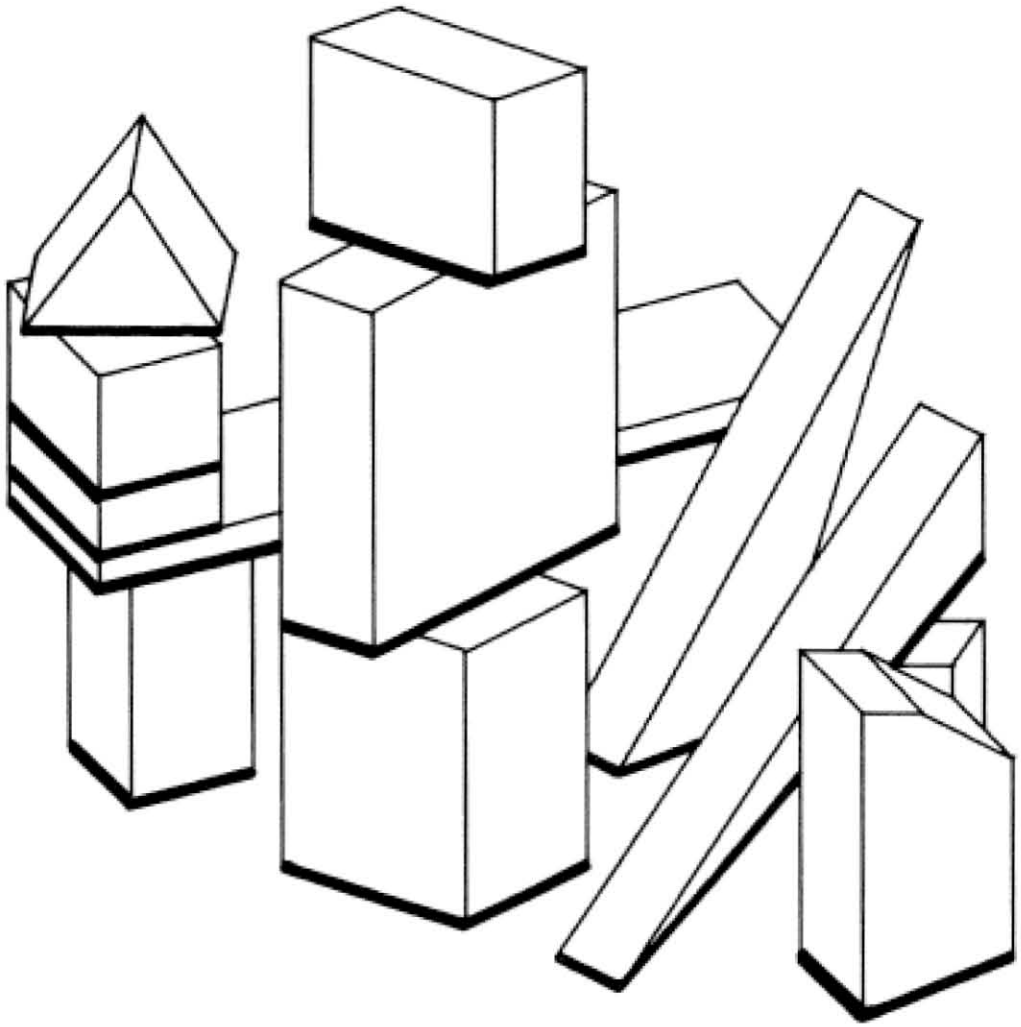


FIGURE 2-13

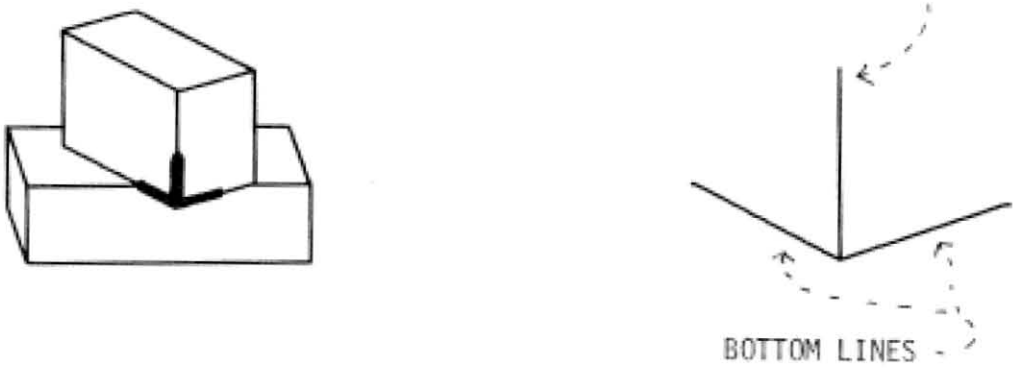
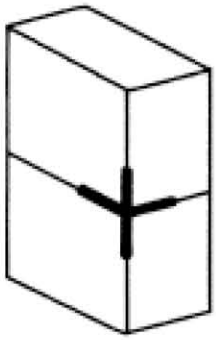


FIGURE 2-14



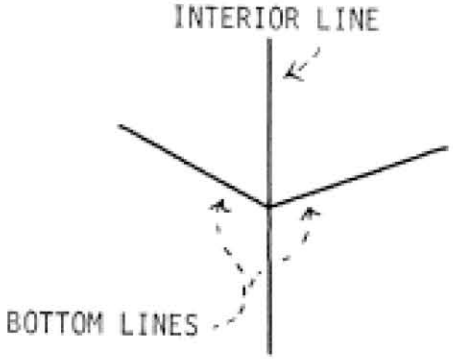


FIGURE 2-15

FIGURE 2-16

This is really a rule and two corollaries, rather than three separate rules. Xs and Ks result primarily when arrows appear incognito, camouflaged by an alignment of objects as illustrated by figure 2-15 and 2-16. Consequently, the corresponding rules amount to locating the arrow-forming parts of the vertex and then acting on that basic arrow.

One further step is necessary before a line can become an approved bottom-line. As shown by figure 2-17, some of the lines qualifying so far must be eliminated. They fail because they are too vertical, or more precisely, because they are too vertical with respect to the arrow's shaft. The effective way to weed out bad lines is as follows:

> Rule: Eliminate any bottom line candidate which is more vertical than the shaft of the arrow suggesting that candidate.

Of course the program extends rudimentary bottom lines through certain vertexes. Figure 2-18 shows the obvious situations in which the bottom line property is extended through the crossbar of a T or the shafts of a pair of matched Ts.

This whole algorithm is based on an assumption that the machine observes the scene from above. If the configuration dangles from the ceiling, simple changes adapt the program to discriminate between UNDER and IN-FRONT-OF, rather than ABOVE

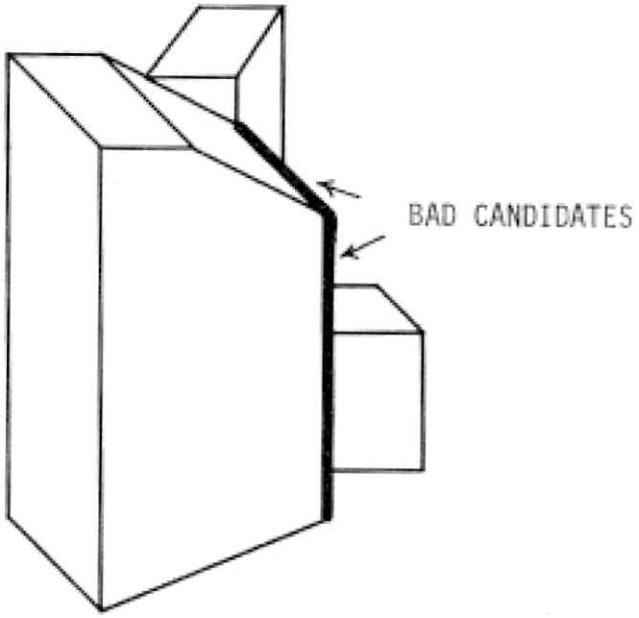


FIGURE 2-17

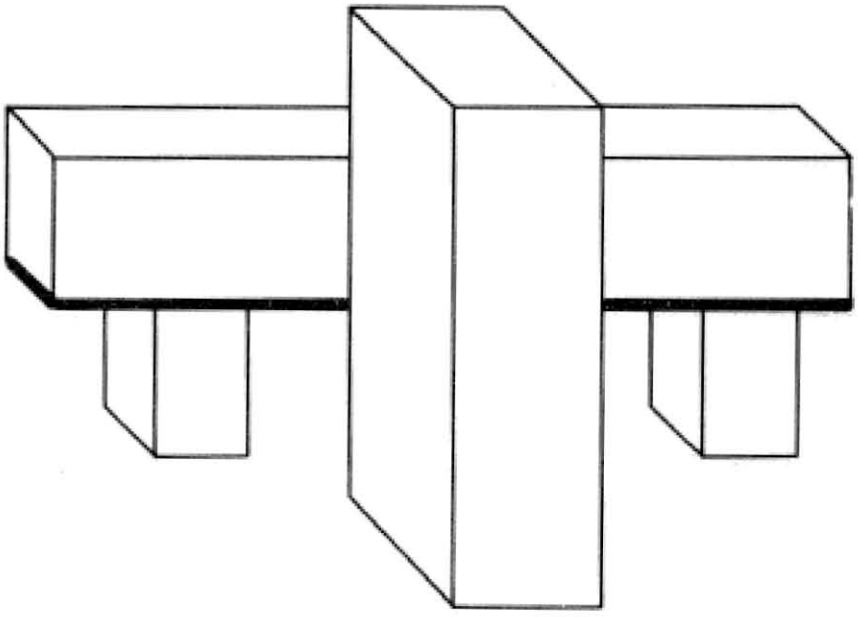


FIGURE 2-18

and IN-FRONT-OF. One examines instead the higher vertexes of the interior lines, substitutes the term top lines for bottom lines in the vertex inspection rules, and the resulting lines usually separate objects from those above them.

> By searching for both the ABOVE and the BELOW relations, the machine may often be able to guess its own height. Consider figure 2-19. Figure 2-20 shows the same scene with top and bottom lines highlighted and with the consequent above and below relations. At least in this case, the machine can correctly deduce that its eye is level with object M because both a chain of above relations and a chain of below relations originate at M.

2.3.1.1 Discussion

This algorithm works effectively because of circumstances all likely but not certain to be true in any particular scene. The method works best when a scene consists of bricks and wedges with one side parallel to the table. In many other cases, the method works anyway, sometimes by coincidence and sometimes by principles not yet fully explored.

> Unfortunately, in explanation I am frequently forced to appeal to intuitive notions about what is likely and what is not. I know of no way to establish a reasonable probability metric on the situations I discuss. All that can be said now is that any such metric should reflect human disposition toward configurations exhibiting alignment and symmetry.

The first likely circumstance or principle is that objects tend to support other objects by contact through relatively horizontal sides. Objects not so supported tend

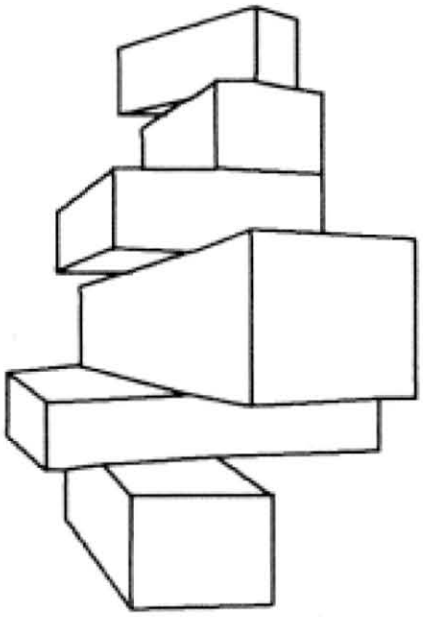


FIGURE 2-19

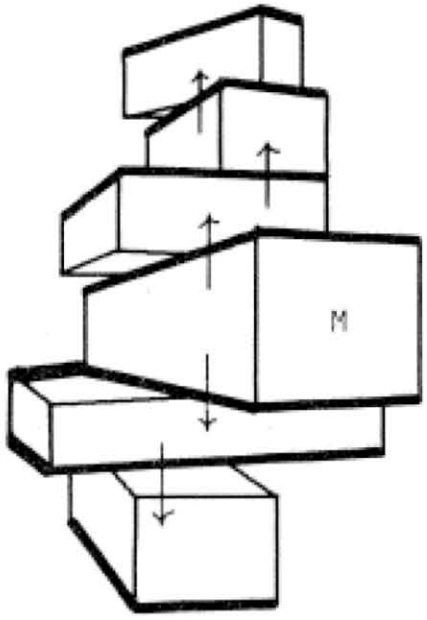


FIGURE 2-20

to slip, although not always as demonstrated by figure 2-21.

Imagine now that the top object in figure 2-22 were completely transparent except for a layer of paint on the crucial, relatively horizontal bottom side. Since the scene is viewed from above, this bared bottom side will obscure part or sometimes all of the supporting object. See figure 2-23. Consequently, some faces of the supporting object generally border on lines resulting from the edges of the bottom side.

Of course many of the bottom side's edges vanish when the object is restored to opacity. Nevertheless, the ones that remain still tend to form part of the seam between the supported and the supporting objects.

Now most of the vertexes of objects are formed by three edges meeting together as in the tip of a pyramid. This forms the so-called trihedral angle. Consequently, when two observable edges of the bottom side form a concave angle, one can expect a third edge of the object to leave the same vertex and form the shaft of a downward directed arrow.

> Slight alteration could permit the program to deal with many objects with non-trihedral vertices. The object in figure 2-24, for example, has two interior lines merging at vertex V. By treating this as a generalized arrow, with multiple shafts, the same algorithm can be used to define bottom lines.

So far the logic is as follows: If one object obscures

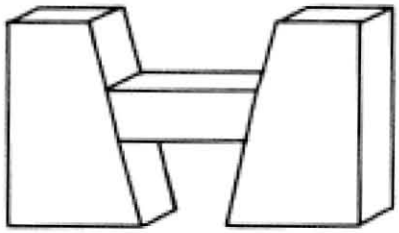


FIGURE 2-21

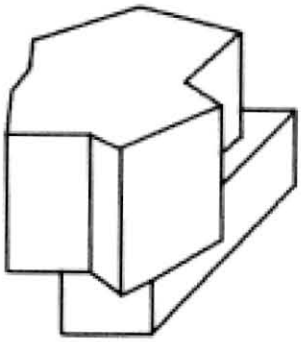


FIGURE 2-22

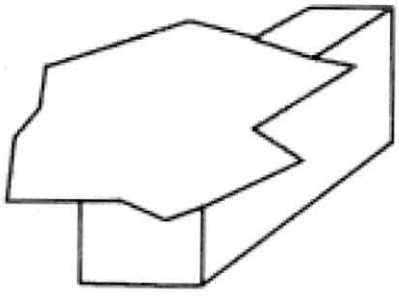


FIGURE 2-23

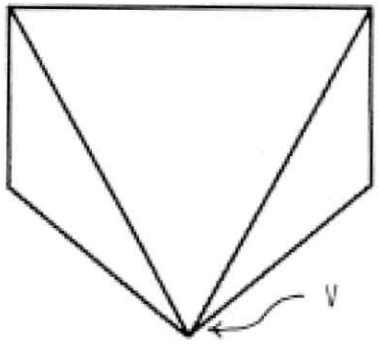


FIGURE 2-24

another because it is on top of the other, then the seam between the two is likely to form the barbs of one or more downward directed arrows belonging to the top object. Reversing this, one would hope for the statement: The barbs of downward directed arrows define seams across which one likely finds the supporting object or objects. Unfortunately one often finds non-supporting objects as well.

2.3.1.2 Refinement

Figure 2-25 suggests a serious kind of over enthusiasm. The pillar, brick B, elevates the bottom lines of brick A and they appear between the viewer's eye and a side of the massive background brick, brick C. In the two brick case, a brick's bottoms lines border on the side of another brick only when the one is in fact supported by the other. When more objects are involved, wrong answers may result because the bottom lines can wander into regions of objects that do not offer support.

To handle this problem, I use a two part procedure. The first part is simply the above algorithm as described so far, which now may be thought of as generating a set of possible supports. The second part, described below, criticizes these possible supports and filters out some of the bad ones.

Now one reason brick A in figure 2-25 clearly does not lie on brick C is that it is absurd to think that an object

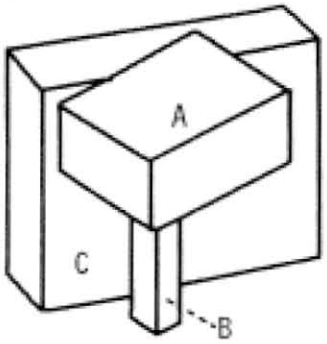


FIGURE 2-25

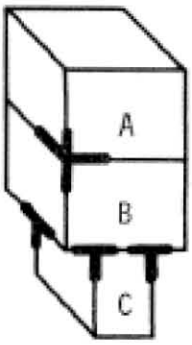


FIGURE 2-26

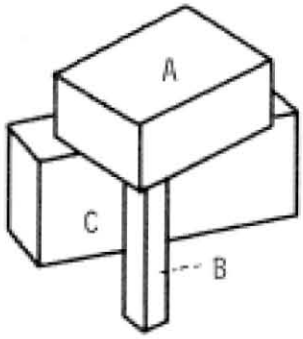


FIGURE 2-27

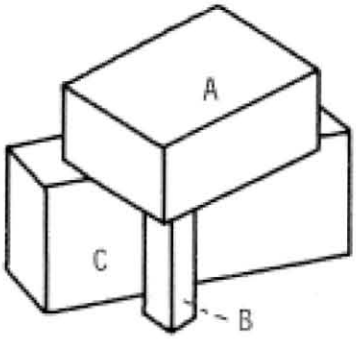


FIGURE 2-28

can rest on a vertical side of some other object. Part 2 of the support algorithm makes sure that just such conjectured supports are eliminated from the list of candidates. To do this, it eliminates a candidate if the only bottom-line-bordering side seems vertical. It assumes a side is vertical if an edge belonging to it is vertical.

There are two exceptions to this rule that occur when an object obscures the entire top of its supporting object as block A obscures block B and as block B obscures block C in figure 2-26. First, if two bricks are aligned as are brick A and brick B, forming the familiar X vertex, no rejection takes place. Second, if the top brick overlaps the support as brick B overlaps brick C, at least one unmatched T appears and again no rejection takes place.

If by this time zero or one support candidate remains, then part 2 terminates, and the support, if any, is announced. There are some common situations, however, that require part 2 to undertake additional computation. Compare figure 2-27 with figure 2-23. The vertical-side filter cannot eliminate the possibility of support from brick C in either figure because one of C's bottom-line bordering sides is clearly not vertical. Yet human observers generally claim the top brick in figure 2-27 cannot be other than singly supported, whereas they admit there may well be support from

the large rear block, C, in figure 2-28.

Since the only difference is in the height of brick B, this judgement must be the result of a height comparision.

Stated simply, the program makes height judgements by assuming an object is supported by the tallest of the support candidates surviving so far. It is simply above the others. The height of a stack cannot always be computed rigorously, however. In simple cases it is sufficient to locate a vertical line belonging to the object and measure it. Brick C in figure 2-27 is such a block. But the verticals of block B disappear into T joints and only minimum heights can be calculated from such lines without complicated and unexplored object extrapolation techniques. Consequently, as the algorithm reviews the heights and minimum heights presented to it, it first selects the maximum of these. Then any candidate whose height is known exactly is rejected if that height is less than the maximum just calculated. All whose exact heights are unknown are allowed to pass.

Figure 2-29 shows why the support algorithm frequently resorts to recursion. If the support for block C is to be calculated, the height of blocks D and E must be compared. But this in turn requires knowledge of their support so that total heights can be added up from the chain of supports and used in this last filtering operation of part 2.

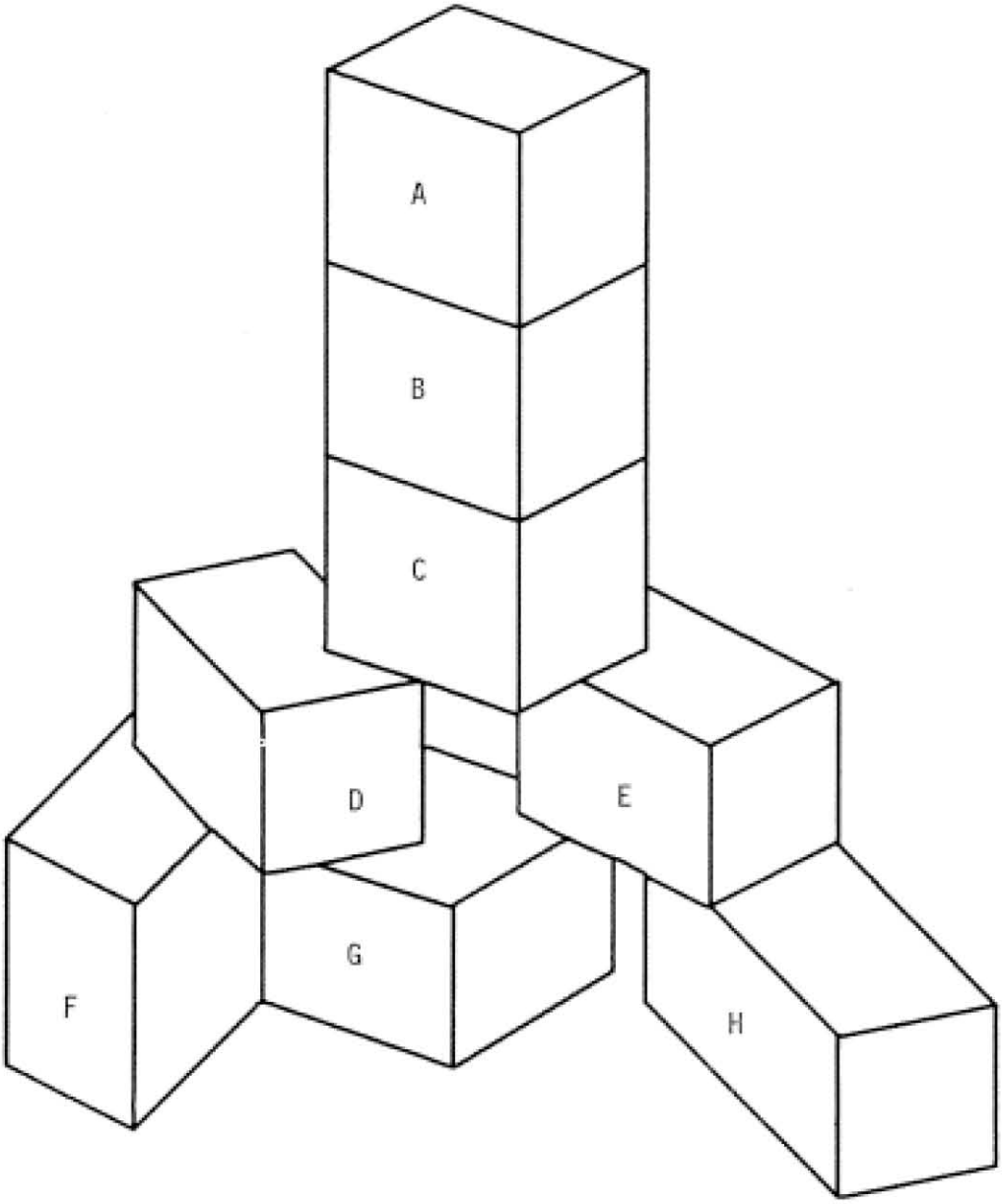


FIGURE 2-29

This completes discussion of the support algorithm as it now stands. It is not hard to delude it deliberately, but it nevertheless operates with a reliability sufficient for use by programs that build upon its results.

Improvements can be made in many ways. The following are a few ideas high on my priority list: 1. A planned modification involves using Ls in the search for bottom lines. So far the system gags on the scene in figure 2-30, finding no bottom lines. But if one line of an L is nearly vertical and the other is nearly horizontal, then the nearly horizontal line should be an excellent bottom line. Of course Ls are frequently buried just as arrows are. Buried Ls are found in certain Ts, Xs and forks as shown in figure 2-31. 2. The discovery of bottom lines can be fouled by introducing small objects that obscure crucial vertexes. Figure 2-32 shows how. This could be corrected by a procedure that extends lines, perhaps after the object is identified. 3. The filtering operation could be strengthened in its ability to detect vertical sides. So far it knows a side is vertical only if the side has a vertical edge. Figure 2-33 shows how it can run aground as a result. None of the sides of brick C appear vertical and the algorithm consequently reports brick A is on top of brick C as well as on brick B.

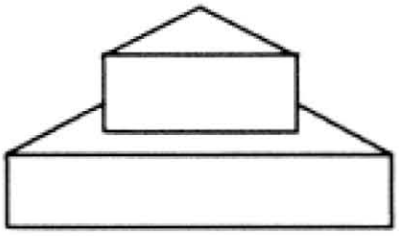


FIGURE 2-30

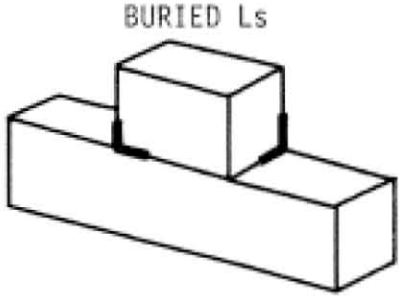


FIGURE 2-31

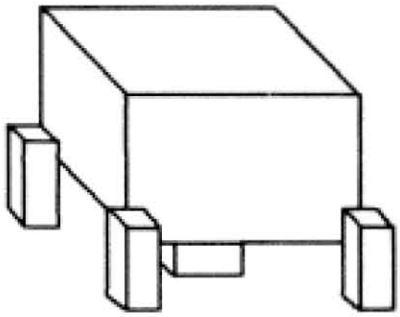


FIGURE 2-32

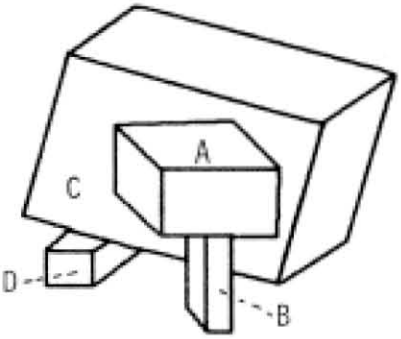


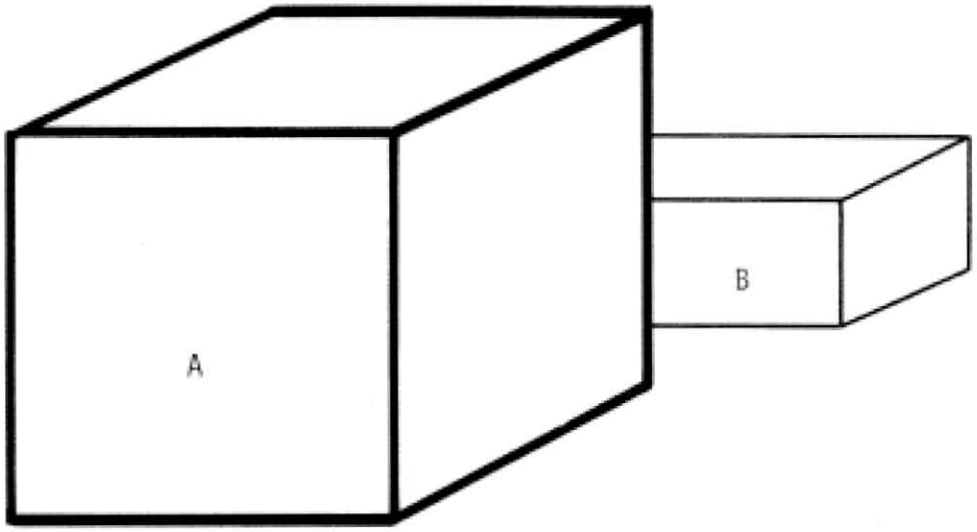
FIGURE 2-33

2.3.2 In-front-of

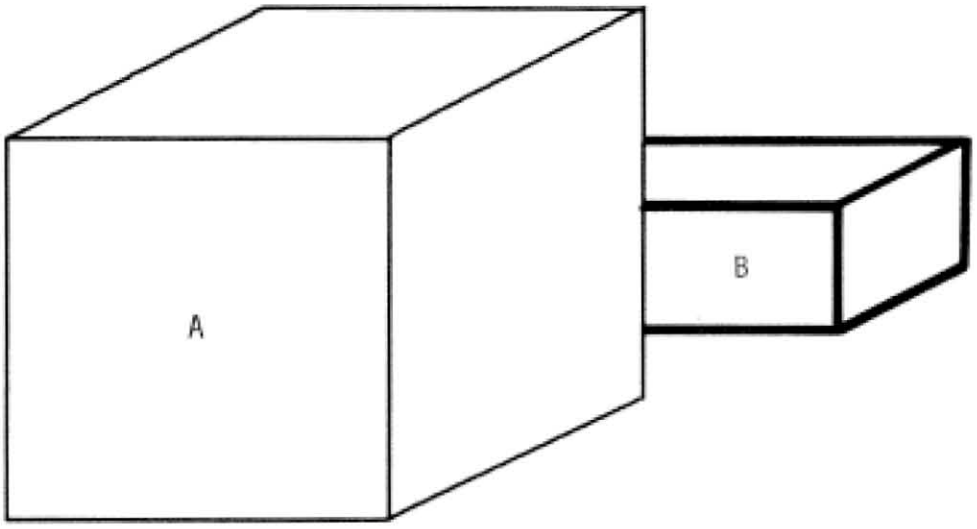
Once a program can discover the SUPPORTED-BY relation, then it can frequently deduce IN-FRONT-OF relations by default. That is, if one of two blocks appears to obscure another but is not above it, then the relation IN-FRONT-OF is a strong possibility. In pursuing this I again use a two part program: the first part proposes possible objects that a given object may be in front of; the second part rejects the bad ones. While simple and direct, this program also succeeds admirably on complex scenes.

Part 1 tries to find all objects that the object in question obscures. First it gathers up most of the obscured objects through search for particular types of T joints on the periphery of the object. Suppose one defines a line to be physically associated with a particular object when that line in the two dimensional drawing results from an edge or intersection of planes on the object, rather than from some obscuring object. Figure 2-34 illustrates. Then the types of Ts sought are just those for which one can be reasonably sure that the crossbar belongs to the object conjectured to be the obscuror.

Figure 2-35 shows two kinds of qualifying T joints. The first kind occurs when the wide angle region associated with the T belongs to or is physically associated with the object



PHYSICALLY ASSOCIATED LINES OF A



PHYSICALLY ASSOCIATED LINES OF B

FIGURE 2-34

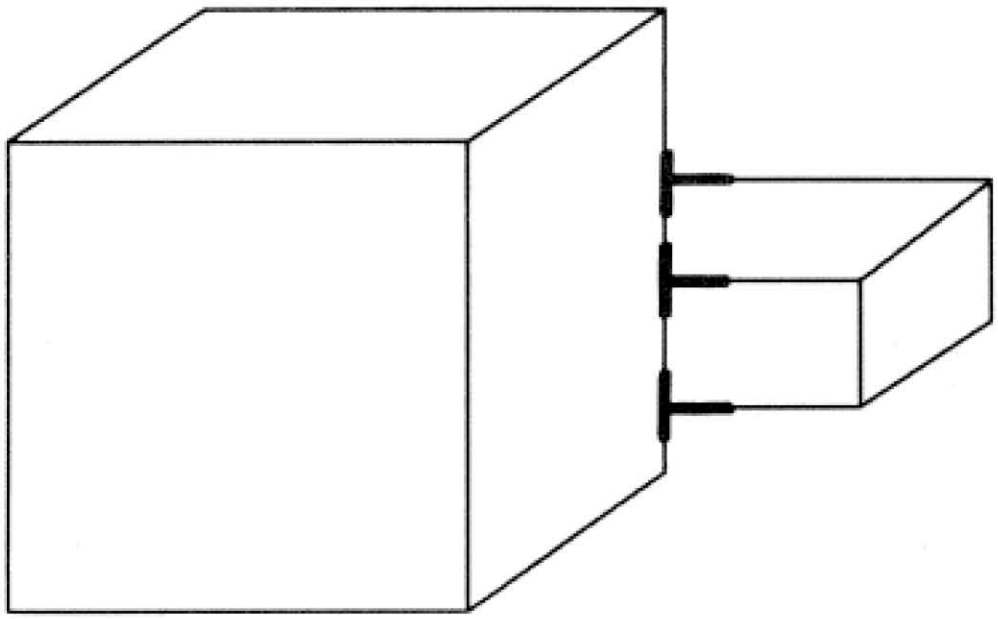


FIGURE 2-35

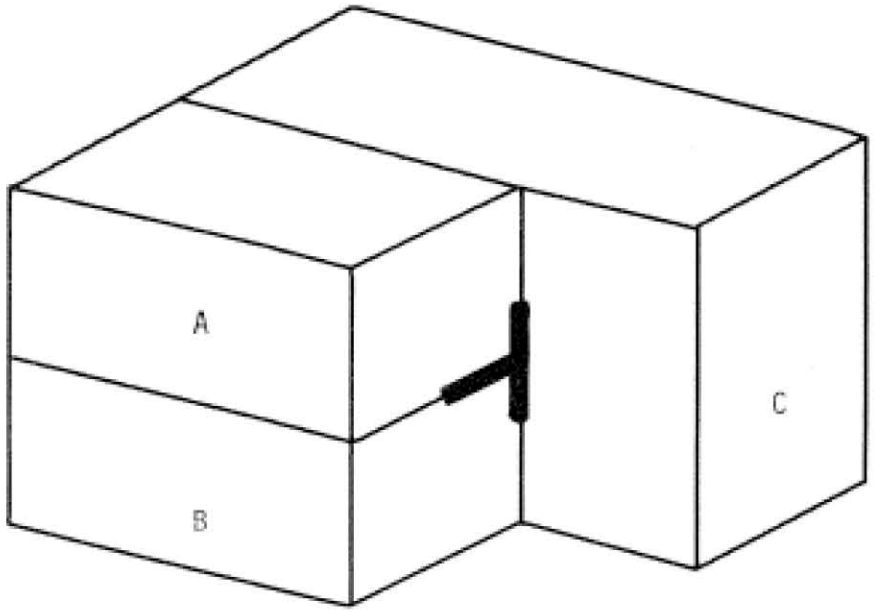


FIGURE 2-36

from which IN-FRONT-OF relations are sought. Both of the other regions, the ones bordering the shaft of the T, belong to a second object. This nearly always indicates the second object is obscured.

The second kind of T joint illustrated in figure 2-35 occurs when one shaft-bordering region belongs to an object while the other belongs to the background. This again assures the machine that the crossbar belongs to the potentially obscuring object.

Figure 2-36 indicates by counter example why nothing can be deduced if the shaft-bordering regions belong to distinct objects. The trouble is that the crossbar of the T is not a real edge of block C, but rather the crossbar is composed of edges belonging to A and B.

Still another way to locate appropriate Ts is more global. The idea is to use whatever means are available to find genuine edges belonging to a body and then to see if any Ts lie along such lines. Recall that the selection of bottom lines involves inspection of arrows, Xs and Ks at the bottom ends of interior lines. Furthermore the rationale behind the support algorithm depends on the likelihood that such bottom lines are physically associated with the same object as the interior line of the arrow, X, or K. Consequently the machine can generate a whole family of peripheral lines likely to be

physical edges by simply examining the arrows, Xs, and Ks at both ends of the interior lines, rather than just those at the bottom ends. Then if any of these physically associated lines end at an L, the other line forming the L is added to the list. It is very unusual for one leg of an L to belong to an object without the other leg belonging also.

Now if any physically associated line is the crossbar of a T, then the parent object obscures some other object or objects. Figure 2-37 demonstrates what kind of edges and Ts are found by this method.

Notice that the Ts referenced by figure 2-37 are also noticed by the previously discussed local inspection since they exhibit the required configuration of objects about the Ts lines. Figure 2-38 demonstrates that both methods contribute, however, since only the local method works on Ts marked L while only the modified bottom-line finder helps on those marked G.

All of this yields obscured objects which are candidates for relating to the object studied by the relation IN-FRONT-OF. Next, part 2 requests help from the support program and then immediately rejects all the candidates for which the ABOVE relation is known to hold. This however is often not completely sufficient. In figure 2-39, the machine knows brick A obscures brick C by virtue of vertex V, But A is not

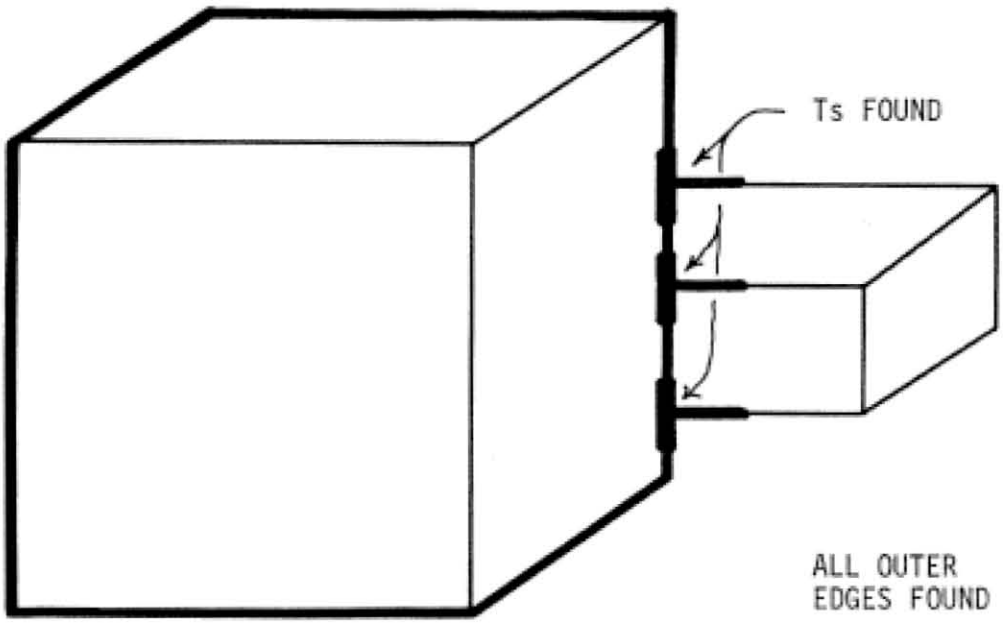


FIGURE 2-37

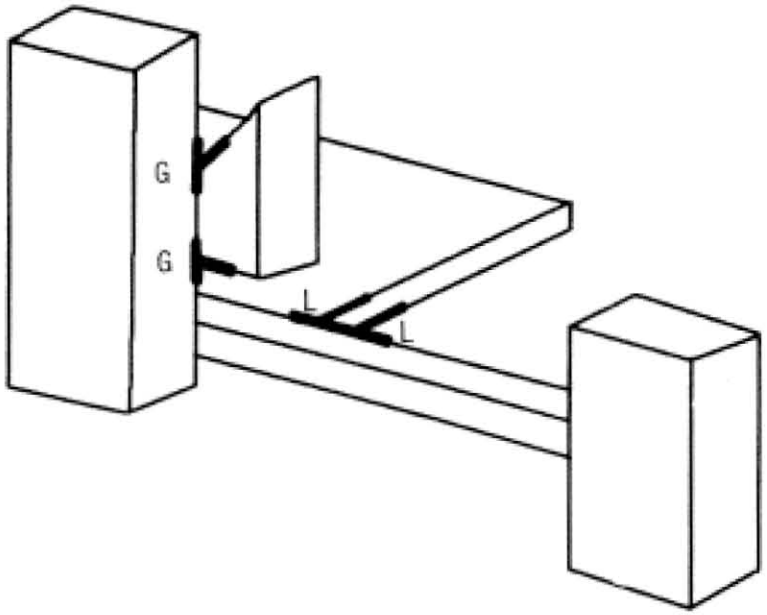


FIGURE 2-38

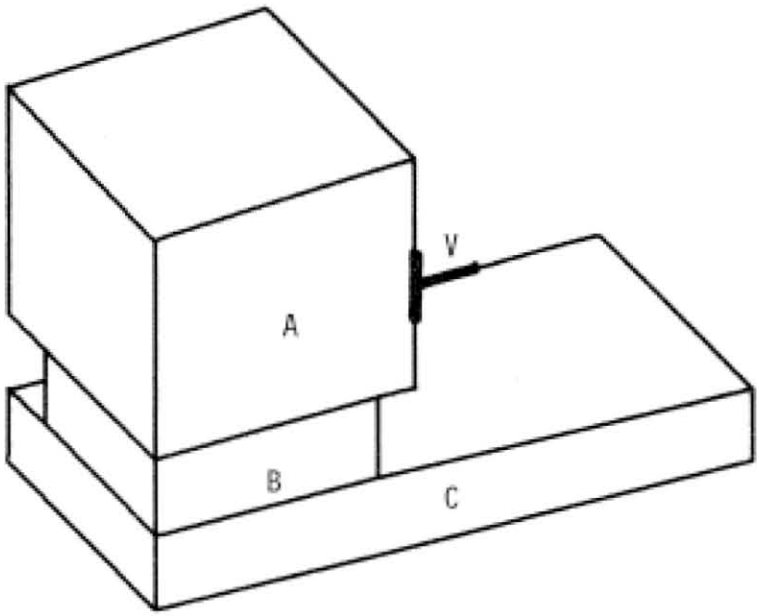


FIGURE 2-39

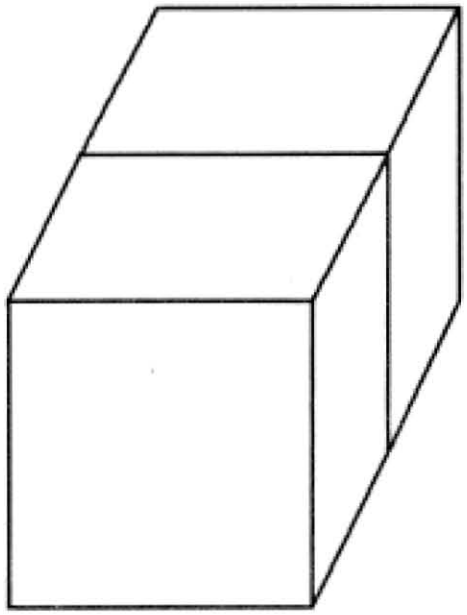


FIGURE 2-40

directly above C. Clearly, the above check must be expanded to the presence of chains of ABOVE relations in order to bring the algorithm into line with human taste. In the course of this check, the program for ABOVE may be called many times if the ABOVE relations have not yet been established. Any candidate that survives this check is thought to be behind the object studied.

The most annoying weakness of this algorithm is that the seam between the obscured and the obscuring object may not exhibit the required type of T joints. Figure 2-40 shows how this can happen. I suspect that further progress can be made in these situations of alignment through close consideration of Xs and perhaps Ks.

2.3.3 An Example

Figure 2-41 provides a somewhat more complex scene for the IN-FRONT-OF and SUPPORTED-BY finding programs to try. The results are as follows:

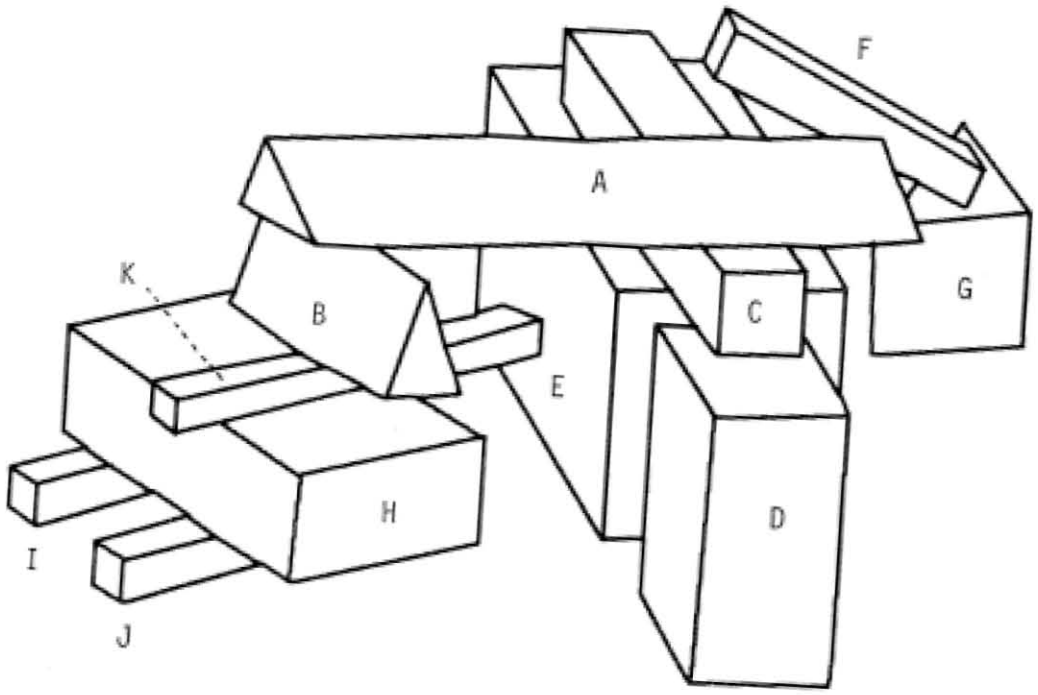


FIGURE 2-41

A	supported-by	B	C	in-front-of	F	G
B		K			-	
C		D	E		-	
D		-			E	
E		-			-	
F		E			-	
G		-			-	
H		I	J		-	
I		-			-	
J		-			-	
K		H			E	

The only bad choice is the neglect of G as a support of F. The reason is that the support criticizing program has a built in assumption that a supported object's bottom is level. Therefore it believes E is the only support for F because it is higher than G, the other possibility.

2.3.4 Left and Right

Two programs exist for deciding if one object is left of, right of, or neither with respect to another. The first computes in a straightforward, simple way. It simply compares the x coordinates of the vertexes of both objects. If there is no overlap, that is if

xcor(any vertex of one object)

<

xcor(any vertex of other object)

then the first object is to the left of the other. If there is overlap, then no statement can be made.

This program, based on the no-overlap criterion, could be greatly improved through the use of an object extending program. The machine is naive to think that object C in figure 2-42 is left of object A. Humans tend to fill in the obscured portions of object C to form a complete block.

But even with an ability to imagine the hidden parts of objects, such a program refuses to really agree with human judgements. Consider the spectrum of situations in figure 2-43. For the first pair of objects, the relations LEFT-OF and RIGHT-OF are clearly appropriate. For the last, they are clearly not appropriate. To me, the crossover point seems to be between the situations expressed by pairs 4 and 5.

Now notice that the center of area of one object is to the left of the left-most point of the other object in those cases where LEFT-OF seems to hold. It is not so positioned if LEFT-OF does not hold. Such a criterion seems in reasonable agreement with intuitive pronouncements for many

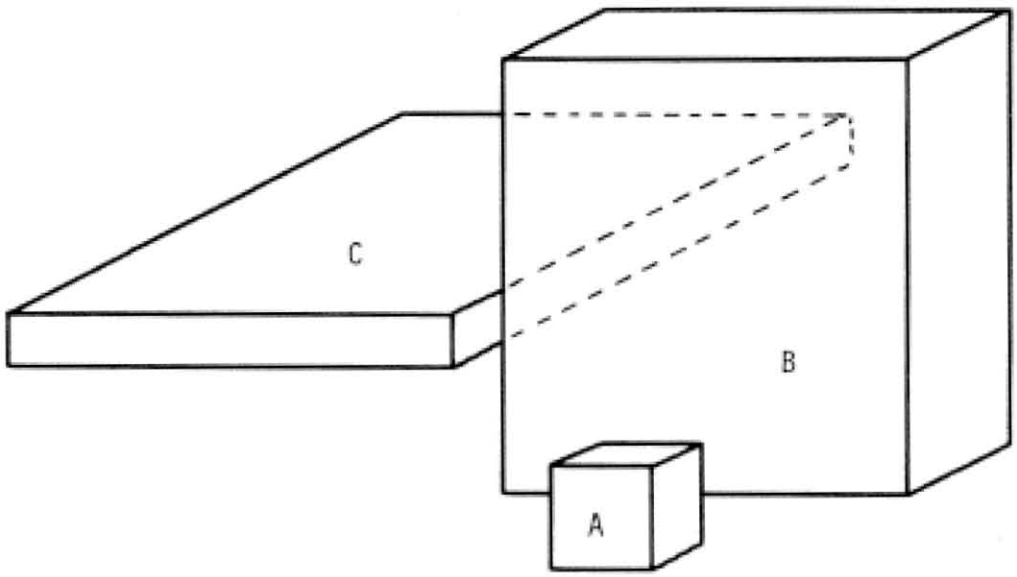


FIGURE 2-42

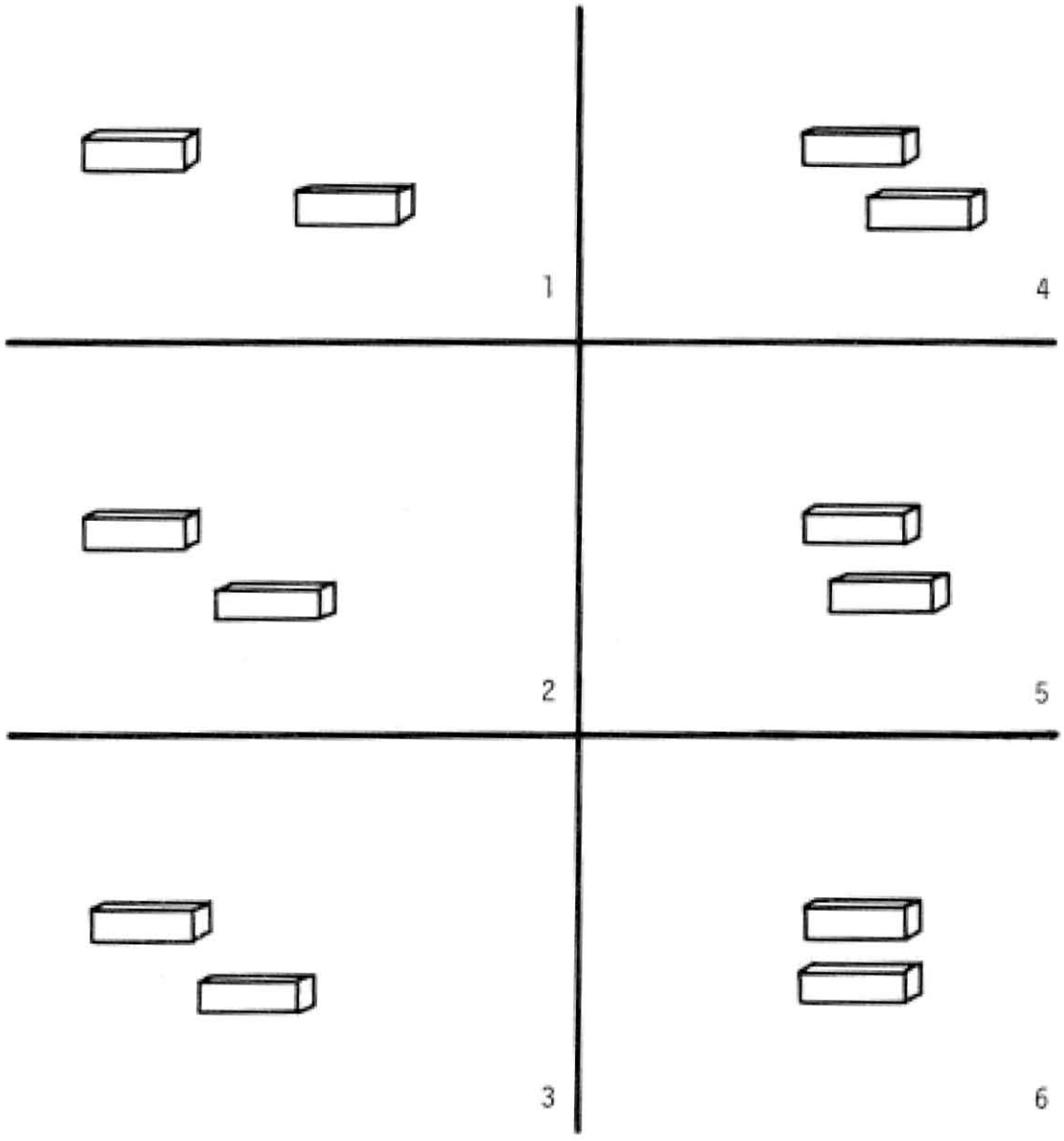


FIGURE 2-43

of the cases I have studied. It also yields reasonable answers in figure 2-44 where in one case A is to the left of B and in the other case it is not. Notice that the relation is not symmetric, however, as the center of area of the much longer brick, brick B, indicates B is to the right of A in both cases.

Figure 2-45 requires extra attention. No matter what the center of mass relations, humans are reluctant to use either LEFT-OF or RIGHT-OF if one object extends beyond the other in both directions. One must additionally specify a rule against this, leaving the following for LEFT-OF:

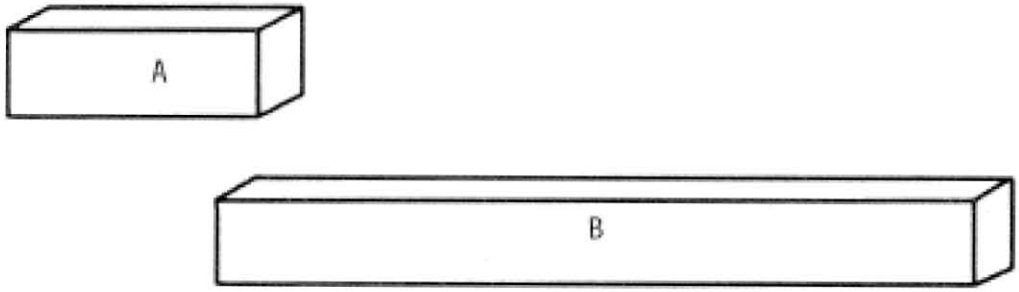
Say A is left of B <==>

1. The center of area of A is left of the leftmost point of B.

2. The rightmost point of A is left of the rightmost point of B.

The rule for RIGHT-OF is of course parallel in form.

> Many people feel their perception of the relation LEFT-OF differs considerably from either of the two possibilities exhibited here. I believe the center-of-area method is reasonable for the machine now, but it would be interesting to more fully explore the question of what humans think to see if other formulas are better. Intuitive notions of LEFT-OF vary wildly and the program can only be said to generally reflect my personal preferences. Indeed, deciding if one object is to the left of another stimulates far more argument than do questions involving relations like IN-FRONT-OF and SUPPORTED-BY. People have difficulty verbalizing how they perceive LEFT-OF and tend to waver in their methods, but implications are that criteria change depending on whether the objects involved are also



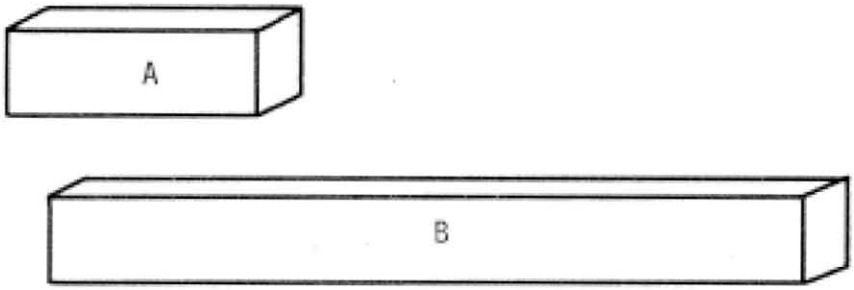


FIGURE 2-44

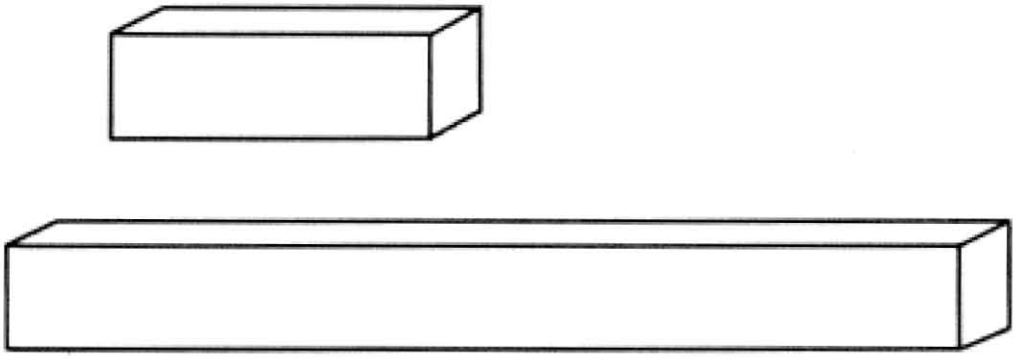


FIGURE 2-45

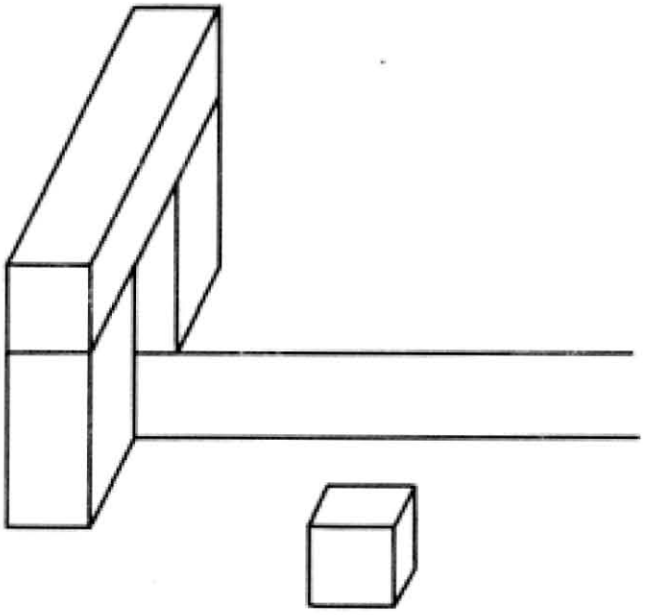


FIGURE 2-46

related by IN-FRONT-OF, ON-TOP-OF, BIGGER-THAN, and so on.

Professor Marvin Minsky has pointed out to me that the orientations of objects are also a strong influence. In figure 2-46, for example, the cube seems left of the arch's entrance even though all its vertexes are clearly right of all the arch's vertexes. In view of this observation, my procedure could probably do better by asking basically the same questions as before, but about lines through the left-most, right-most, and center-of-area points in the direction of orientation instead of what amounts to vertical projection of the points to the x-axis. But then there is the problem of finding an object's intrinsic orientation. At the moment I know of no general heuristics for this.

2.3.5 Marrys

The abuts and aligned-with relations arise frequently, perhaps because of some human predilection to order. As intuitively used, however, neither of these words corresponds to the notion I want the machine to deal with. To avoid confusion, I therefore prefer to use the term marry which I define as follows:

> Definition: An object marrys another if those objects have faces that touch each other and have at least one common edge.

Thus the objects in figure 2-47 are said to marry one another. Those in figure 2-48 do not because they have no common edge. Similarly those in figure 2-49 do not because they have no touching faces.

The MARRYS relation is sensed by methods resembling those previously described. First the vertexes along the border are collected. Then the Xs, Ts and Ks are further

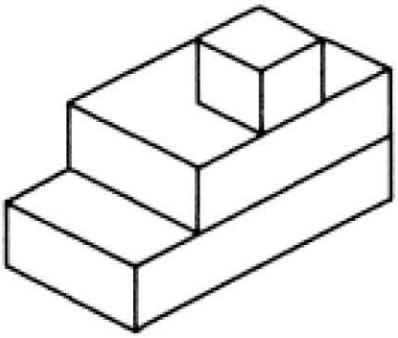


FIGURE 2-47

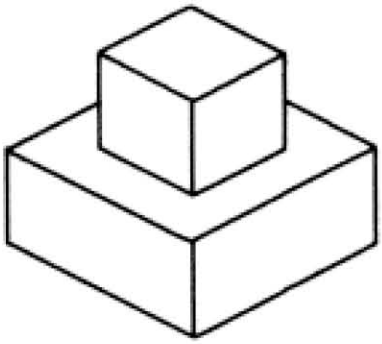


FIGURE 2-48

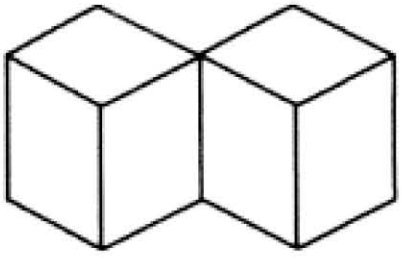


FIGURE 2-49

examined:

The Xs are simple to handle. If exactly two of the lines are collinear and if the other two separate two objects, then those objects are very likely to deserve the MARRYS relationship because such a vertex is strongly associated with aligned stacks or rows. Figure 2-50 illustrates these situations. Figure 2-51 shows why the two objects must be separated by the two non-collinear lines. There three sides belong to the same object and the MARRYS relation does not hold.

If there are four objects at the vertex and there are two pairs of collinear lines, then the likely situation is a field of objects with those sharing lines marrying each other. See figure 2-52.

Ks are strongly correlated with the sort of alignment illustrated by figure 2-53. The rule is simple: If there are two objects with sides at a K vertex, then they probably marry.

When three objects meet at a T one of the following holds:

1. The objects with the stem in between marry each other and both obscure the third object.
2. The third object is the obscuring object. In this case the two other objects may or may not marry.

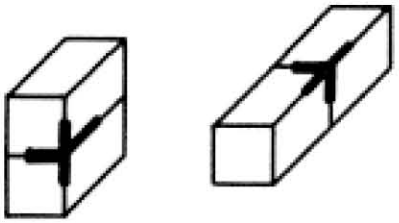


FIGURE 2-50

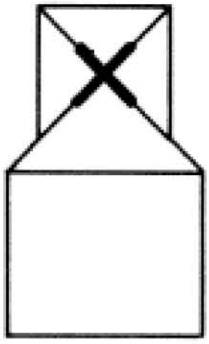


FIGURE 2-51

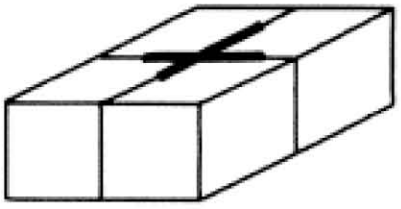


FIGURE 2-52

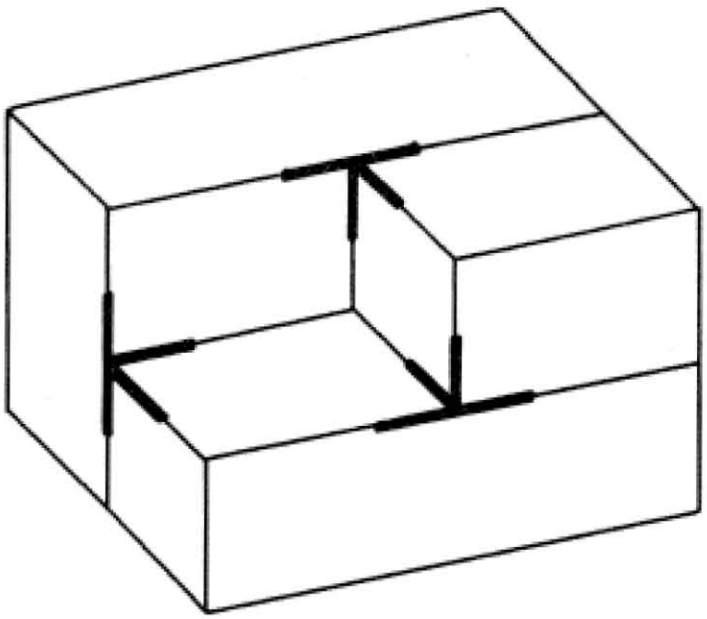


FIGURE 2-53

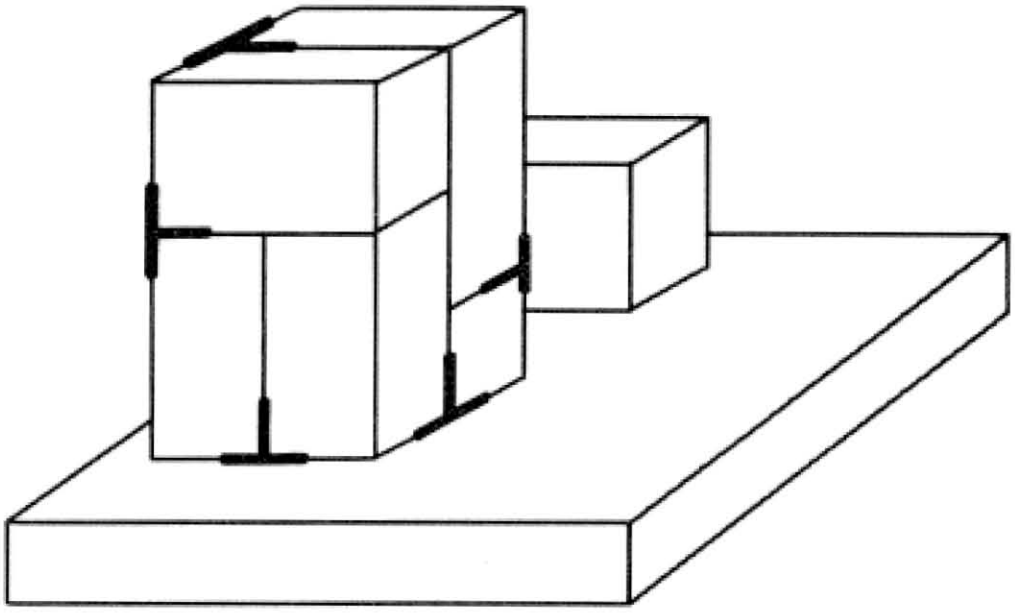


FIGURE 2-54

3. All three objects marry.

4. Something else.

Of these, my programs check for case 1 only. The central program looks for chains of IN-FRONT-OF and SUPPORTS relations between the shaft-bordering objects and the large-angle object. If such chains are found for both, they likely both obscure the large angle side and marry each other. Figure 2-54 shows examples.

Now consider the situation where only two objects meet at a T with the wide angle side and one of the other sides belonging to one object. As figure 2-55 suggests, this sort of T frequently becomes a K when seen from some other angle. Like the K, the machine considers it strong enough evidence for a MARRYS relation.

Figure 2-56 illustrates both of the T joint situations that confirm marrymen t.

Note that the machine is conservative in using this MARRYS relation; the relation is not placed in ambiguous situations such as those of figure 2-57.

2.3.6 Shape

Before an object can be identified, the lines that are really edges of that object must be sorted from those that are edges of obscuring objects. It would not do to think object B in figure 2-58 has sides shaped like those in

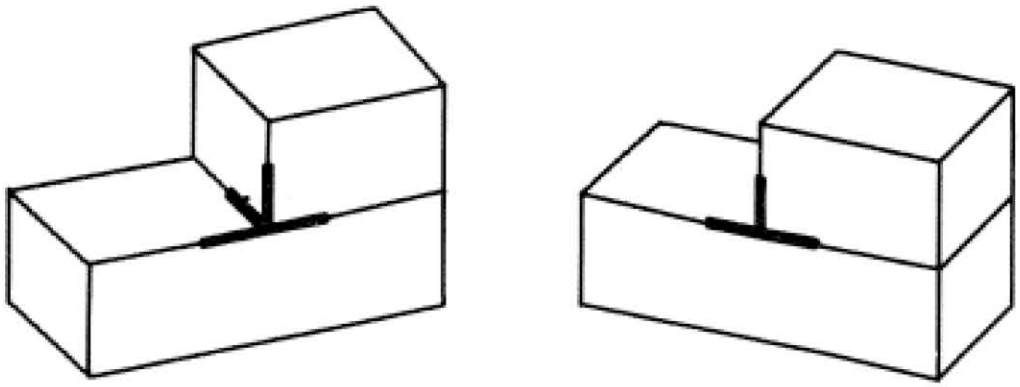


FIGURE 2-55

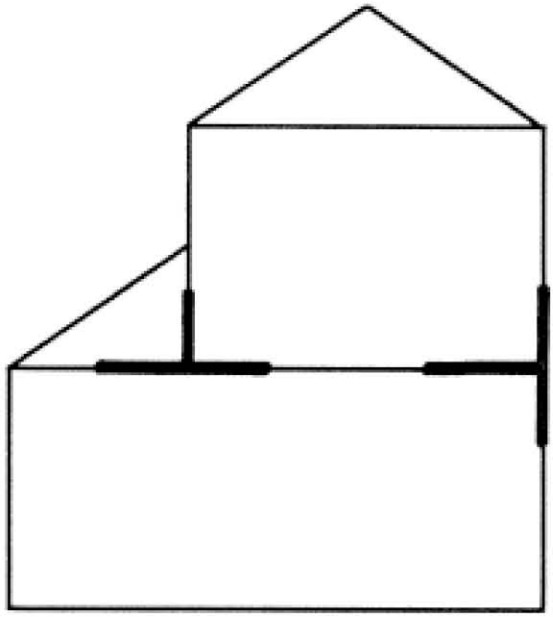


FIGURE 2-56

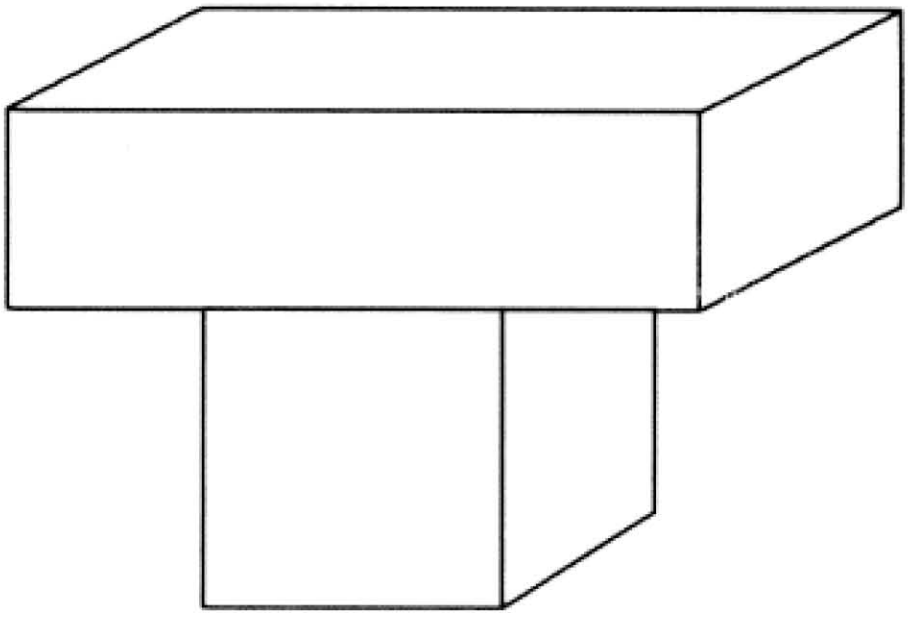


FIGURE 2-57

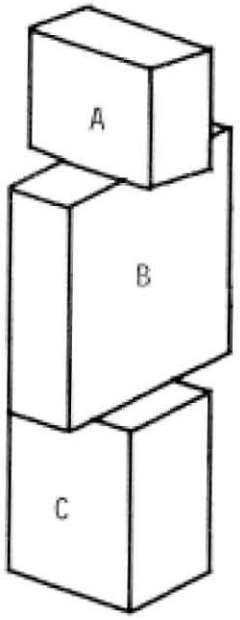


FIGURE 2-58

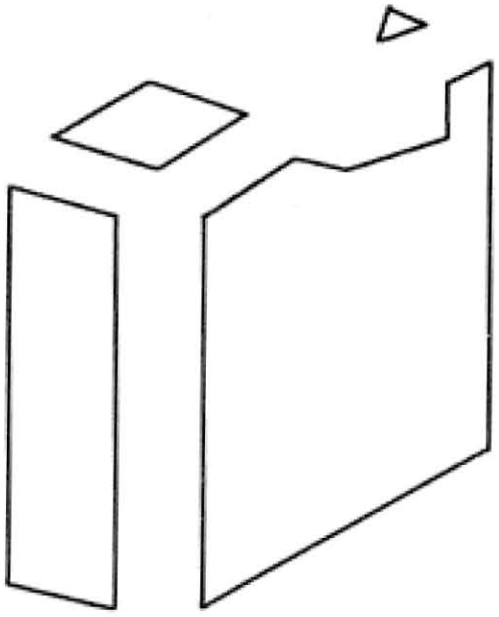


FIGURE 2-59

figure 2-59.

The idea of an edge belonging to a body has been discussed. The shape program I use first gathers together those lines found to be genuine physically associated edges by the program that looks for IN-FRONT-OF relations. To these it adds any lines that lie between two regions of a body, the interior lines. Then if these lines include both uprights in a pair of matched Ts, it adds a line joining the two Ts. And finally, any line shared with the background or other body known to be below or behind is certainly included. These rules are sufficient to identify many of the lines that belong to any given object, while rejecting many that do not belong.

Figure 2-60 shows how this program sees object B of figure 2-58. Lines L, M, N, and O are interior lines. Line P is a segment between matched Ts with the required kind of uprights. Q qualifies by way of the IN-FRONT-OF algorithm, while R, S, T, U, and V qualify both by way of the IN-FRONT-OF algorithm and the rule adding lines lying between the object and the background. Figure 2-61 shows how the rest of the scene in figure 2-58 is dissected by this program. Notice that the shapes are reasonably well defined.

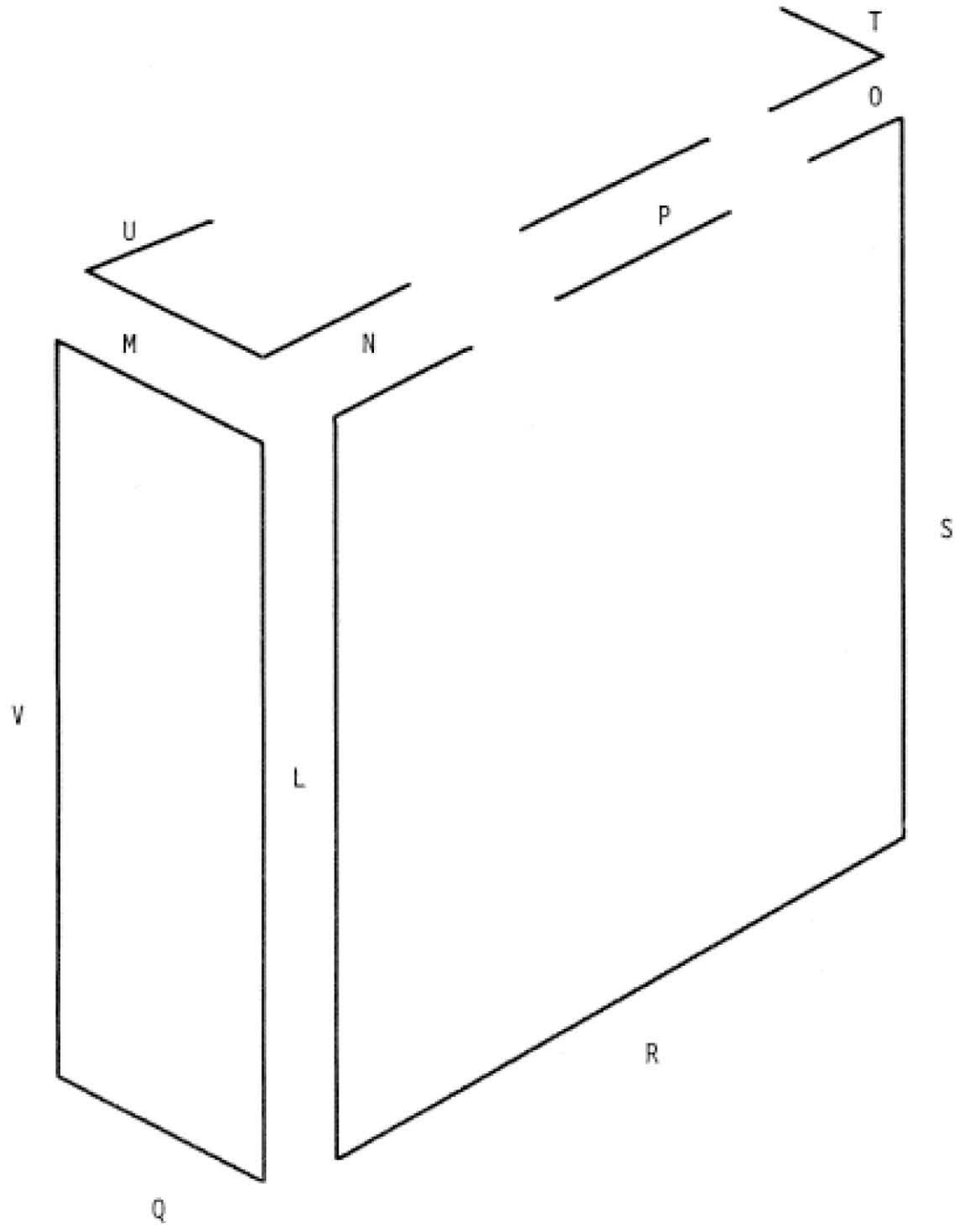


FIGURE 2-60

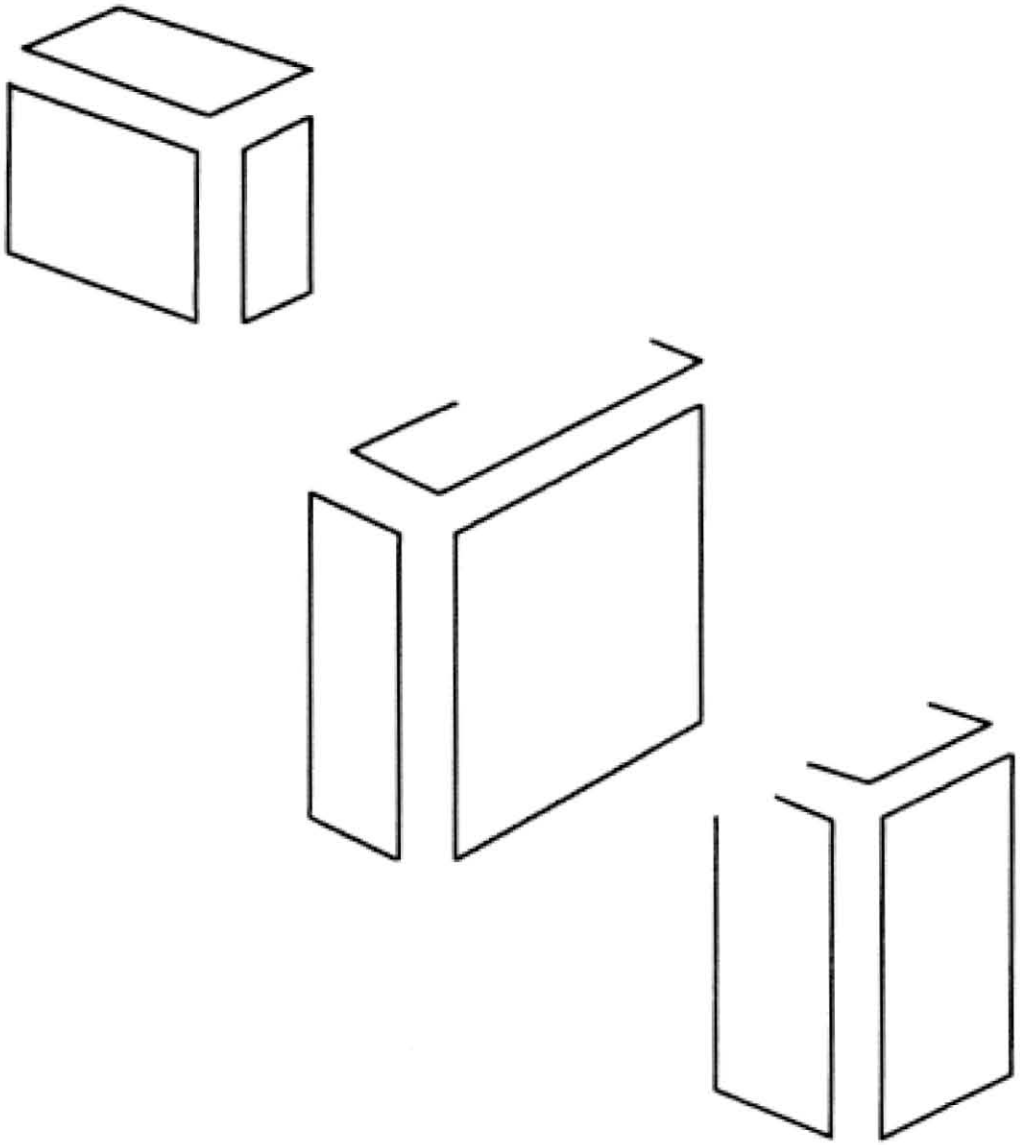


FIGURE 2-61

2.3.7 Size

Piaget has shown that at a certain age children generally associate physical size with greatest dimension [3]. They will, for example, adamantly maintain that a tall thin beaker has more water in it than a short fat one even though they have seen them filled from other beakers of equal size.

Adults do not develop as far beyond this as might be expected. I do not think we really use the notion of volume naturally. Apparent area seems much more closely related to adult size judgement. Notice that beaker A in figure 2-62 appears to have about the same amount of water in it as does beaker B, even though it must contain twice as much. Unless a subject consciously exercises a formula for volume, he is likely to report that object B in figure 2-63 is approximately ten times larger than object A, even if told both objects are cubes. The true factor of twenty-seven times seems large when the trouble is taken to calculate it.

Consequently, the size generating program does not trouble with volume. Instead it calculates the area of each shape produced by the shape detecting algorithm. Next it adds together the areas of all shapes belonging to an object to get its total area. Then using these areas it can compare two objects in size or consult the following table for a

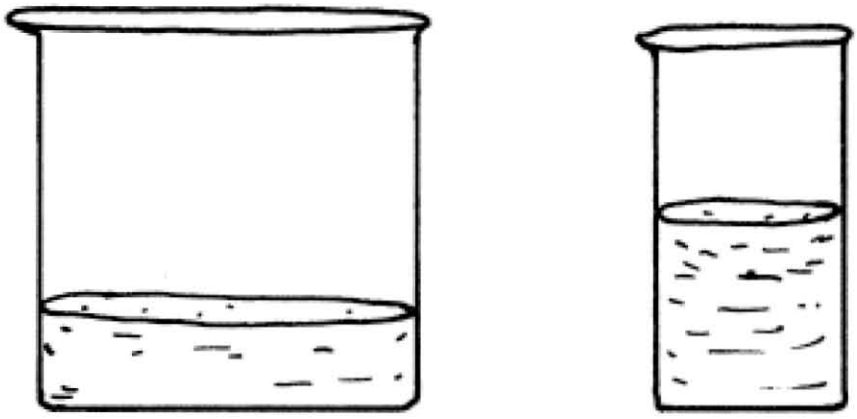


FIGURE 2-62

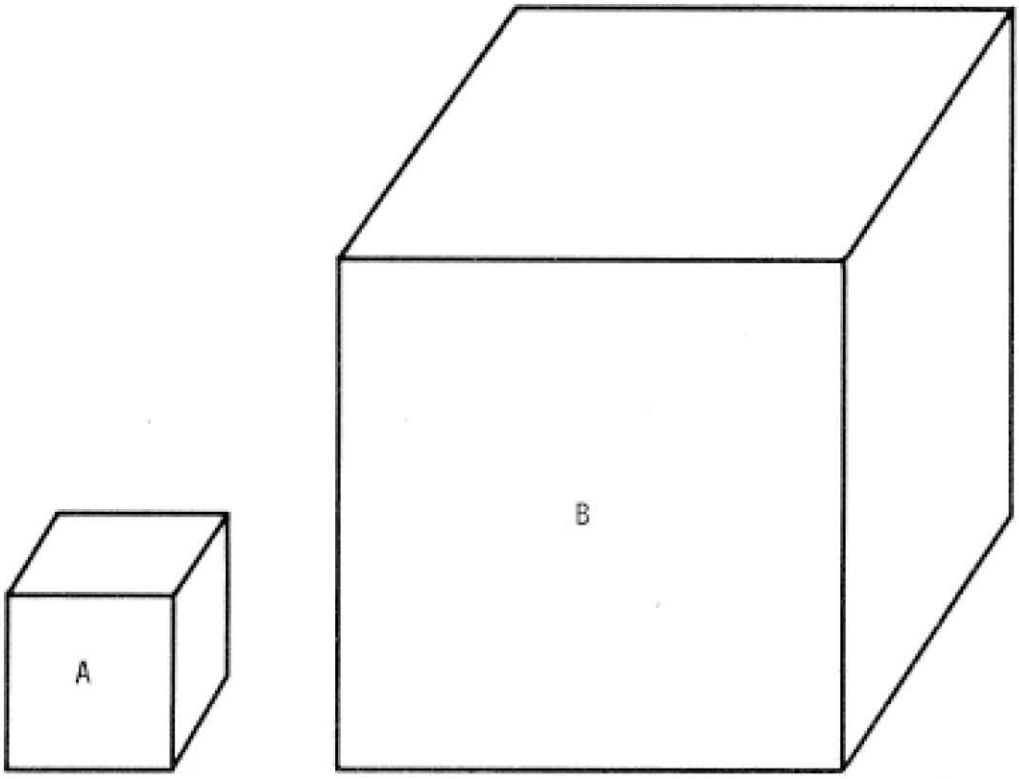


FIGURE 2-63

reasonably believable discrete partitioning of the area scale:

0.0%	to	0.5%	of the visual area	-->	tiny
0.5%	to	1.5%	of the visual area	-->	small
1.5%	to	15%	of the visual area	-->	medium
15%	to	35%	of the visual area	-->	large
35%	to	100%	of the visual area	-->	huge

3 Discovering Groups of Objects

When a scene has more than a few objects, it is usually useful to deepen the heirarchy of the description by dividing the objects into smaller groups which can be described and thought of as individual concepts. Figure 3-1, for example, seems to divide naturally into three groups of objects, one being three objects tied together by SUPPORTED-BY pointers, another being three similar objects on top of a fourth, and the third being a set of objects in the arch configuration. There are other kinds of grouping humans use, but in this work I primarily explore only the three illustrated by this figure 3-1. Grouping by identification with a known model is discussed later in the chapter on identification. This chapter deals with grouping on the basis of pointer chains and on the basis of property similarities.

3.1 Sequences

A simple kind of group consists of chains of SUPPORTED-BY or IN-FRONT-OF pointers as in the tower of figure 3-1. The first act of the grouping program is to find sets of objects that are so chained together. All such sets with three or more elements qualify as groups.

In the event the sequence of pointers closes on itself, a ring is formed. In figure 3-2 there is such a group because each of the three objects rests partly on one of the

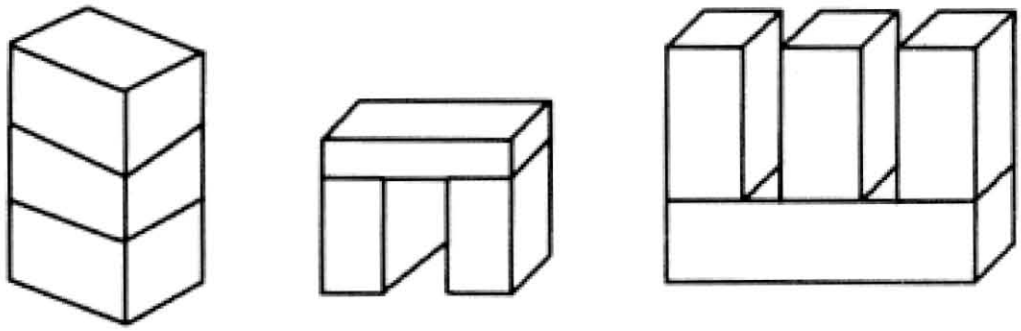


FIGURE 3-1

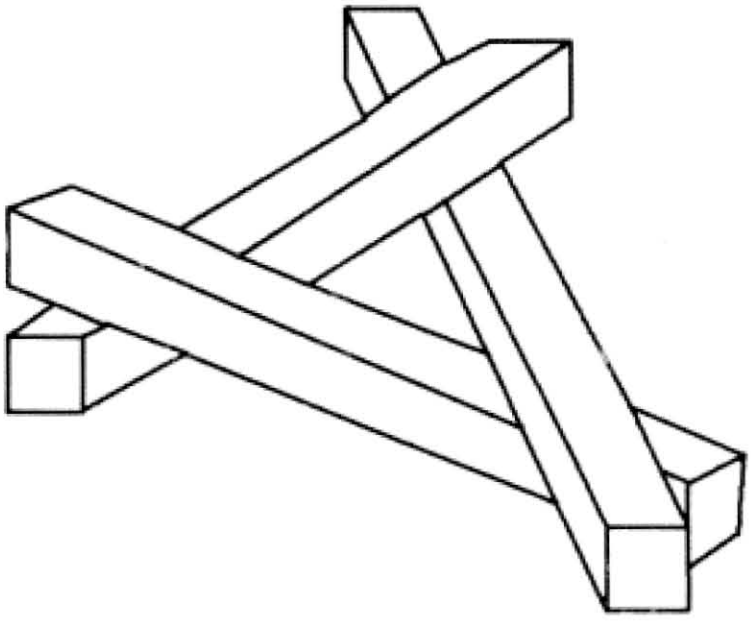


FIGURE 3-2

other two. The result is a circular chain of SUPPORTED-BY pointers as shown in figure 3-3.

Using chains to define groups can become fairly complex as illustrated by the scene in figure 3-4. A chain of SUPPORTED-BY pointers splits into two branches in scene one at the point where object C is supported by two objects, D and E. In scene two, two chains of SUPPORTED-BY pointers join at M which supports both I and L. The current version of the grouping program terminates chains at junction points without further fuss. This seems reasonable for it seems natural to think of the scenes in figure 3-4 as a set of groups consisting of A-B-C, G-H-I, and J-K-L.

> Another kind of problem arises when objects tied together by a simple chain of relations should not constitute a group because of other factors. Figure 3-5 shows one kind of situation that can occur, for which I have only ideas but no programs. In this scene the machine perceives a single object conglomerate, grouped together by virtue of an unbroken chain of SUPPORTED-BY pointers. But most humans see a short tower on top of a board on top of another tower. This must be partly because of the size differences and partly because of the fact that the top group is not directly over the other objects. In any case, it would seem that radical change in object properties should be possible grounds for breaking a chain. With this, one is into territory where irrevocable committments should be avoided. Perhaps the best thing would be to have the grouping program offer alternative groupings of tricky scenes and postpone decision until higher level identification programs indicate which arrangement leads to the best match of the scene against known models.

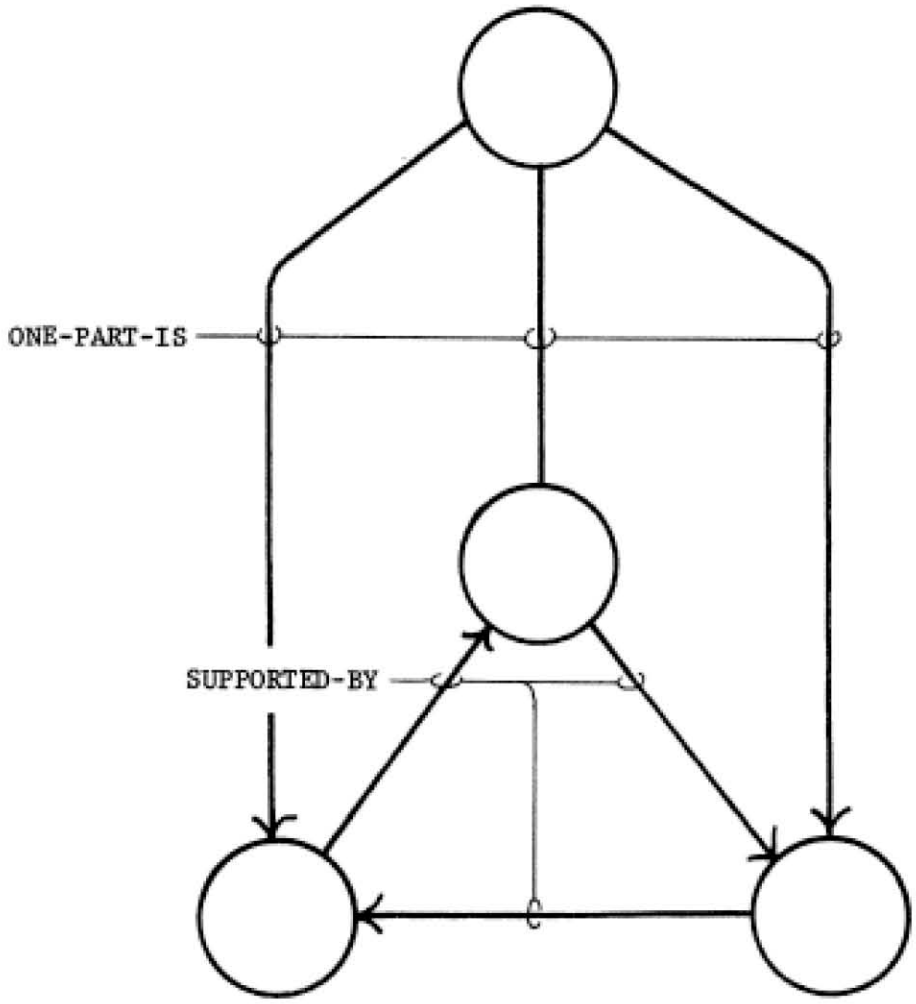


FIGURE 3-3

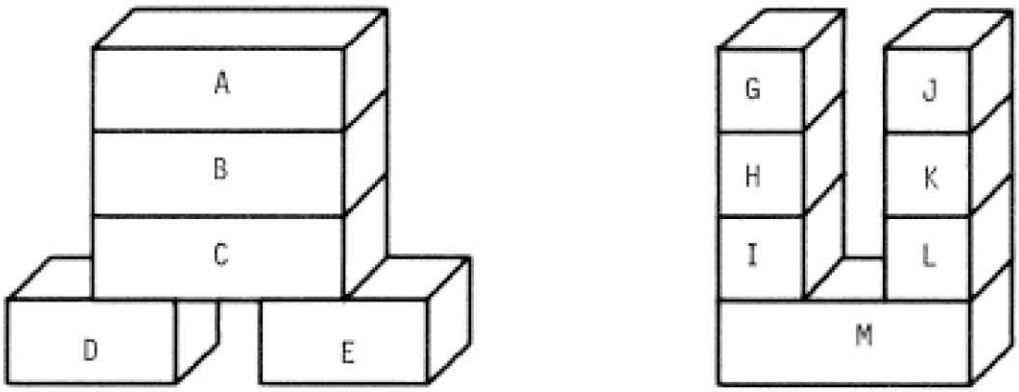


FIGURE 3-4

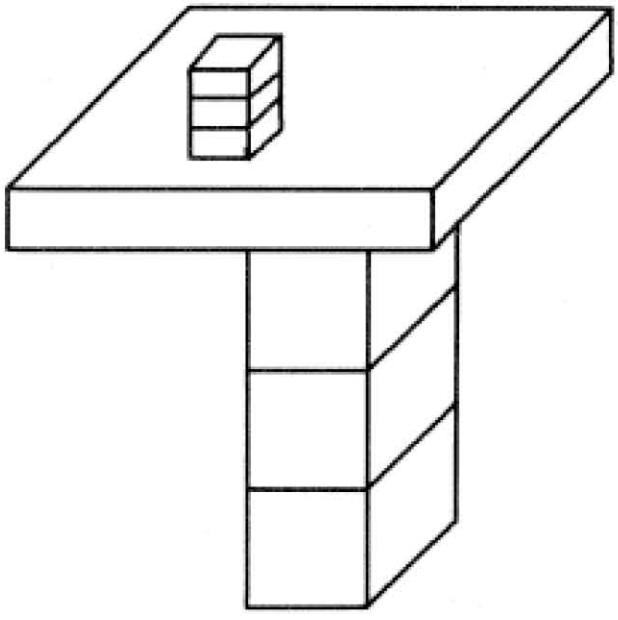


FIGURE 3-5

3.2 Common Properties

When several objects have the same or very nearly the same description, they are immediately solid candidates for a group. The legs on the table in figure 3-6 are typical. All are bricks, all are standing, and all are supports for the top board.

This kind of manipulation is slightly dangerous in that my criteria for forming a group and admitting members to it are a bit flimsy. So far the rules are based on the following demands:

1. All candidates for group membership must be related to one or more particular objects in the same way. For the table case, all four objects are related to the board by SUPPORTED-BY. This restriction appears necessary because uniform relationship to a single object seems to have strong binding power. The standing bricks in figure 3-7 naturally constitute two groups, not one.

2. There must be three or more members in the group, and the members of the group must share many of their properties.

Figure 3-8 outlines the procedure for forming such groups. The basic idea is to make a generous guess as to what objects to include in a group and then to eliminate objects which seem atypical until a fairly homogeneous set remains.

To do this, a program first finds a candidate group by locating a set of objects that relate to one particular

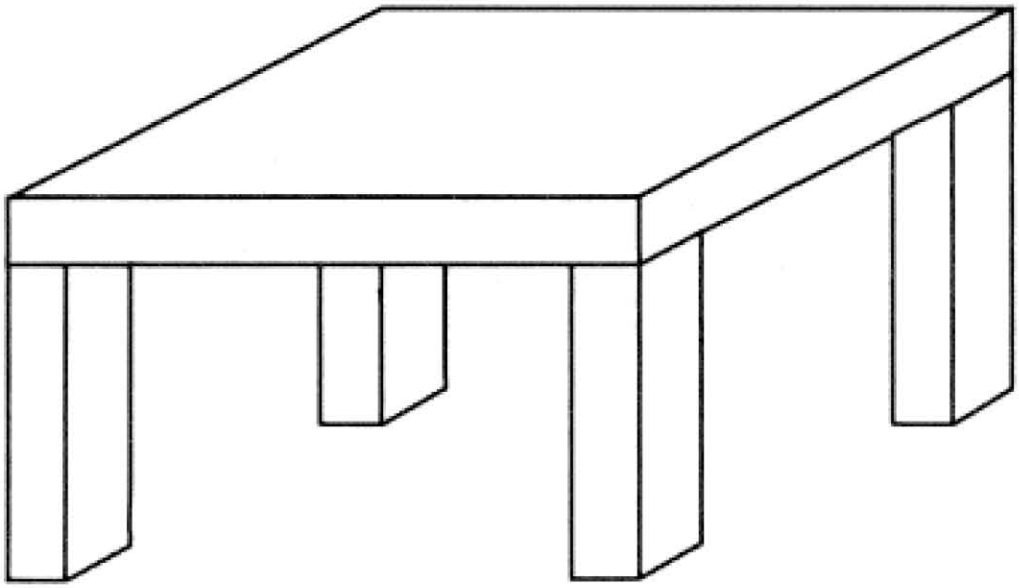


FIGURE 3-6

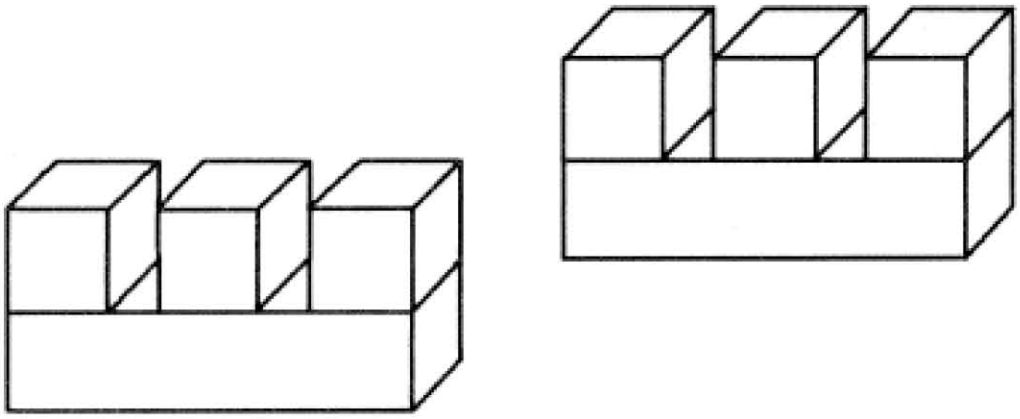


FIGURE 3-7

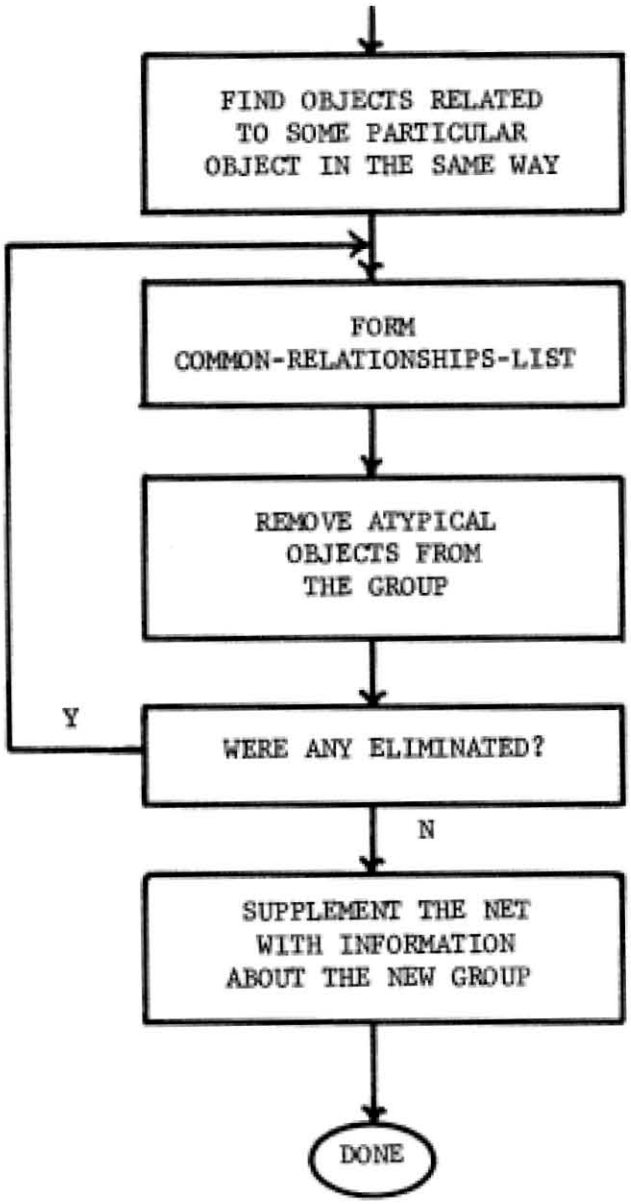

FIND OBJECTS RELATED TO SOME PARTICULAR OBJECT IN THE SAME WAY
FORM COMMON-RELATIONSHIPS-LIST
REMOVE ATYPICAL OBJECTS FROM THE GROUP
WERE ANY ELIMINATED?
Y
N
SUPPLEMENT THE NET WITH INFORMATION ABOUT THE NEW GROUP
DONE

FIGURE 3-8

object in the same way. Next comes formation of a common-relationships-list through a listing of all relationships exhibited by more than half of the candidates in the set.

Figure 3-9 helps explain this process. Objects A through F are immediately perceived to be a possible group because they all have a relationship, SUPPORTED-BY, with a single object, G. The relationships exhibited by the candidates are:

A, B, and C:

1 SUPPORTED-BY pointer to G
2 MARRYS pointer to G
3 A-KIND-OF pointer to BRICK
4 HAS-PROPERTY-OF pointer to MEDIUM-SIZE

D:

1 SUPPORTED-BY pointer to G
2 MARRYS pointer to G
3 A-KIND-OF pointer to BRICK
4 HAS-PROPERTY-OF pointer to SMALL

E and F:

1 SUPPORTED-BY pointer to G
2 MARRYS pointer to G
3 A-KIND-OF pointer to WEDGE
4 HAS-PROPERTY-OF pointer to SMALL

Three relations appear in the common-relationships-list because they are found in more than half of the candidates' relationships lists:

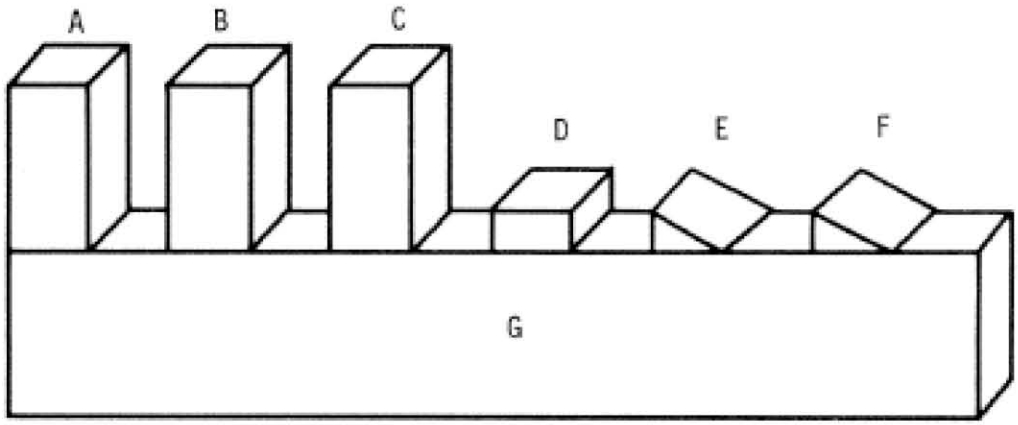


FIGURE 3-9

common-relationships-list:

1 SUPPORTED-BY pointer to G
2 MARRYS pointer to G
3 A-KIND-OF pointer to BRICK

After this common-relationships-list is formed, all candidates are next compared with it to see how typical each is. The measure is simply the fraction of the total number of properties of the candidate and the common-relationships-list that are shared. Said in a more formal way, the measure is

$$\frac{\text{number of properties in intersection}}{\text{number of properties in union}}$$

where the union and intersection are of the candidate's relationships list and the common-relationships-list.

Figure 3-10 represents abstractly a situation in which the candidate and the common-relationships-list are quite different. The shared properties, represented by the shaded area, is but a very small fraction of the total area, both shaded and unshaded. Figure 3-11 gives the opposite extreme. There is considerable overlap and the value is near one, the

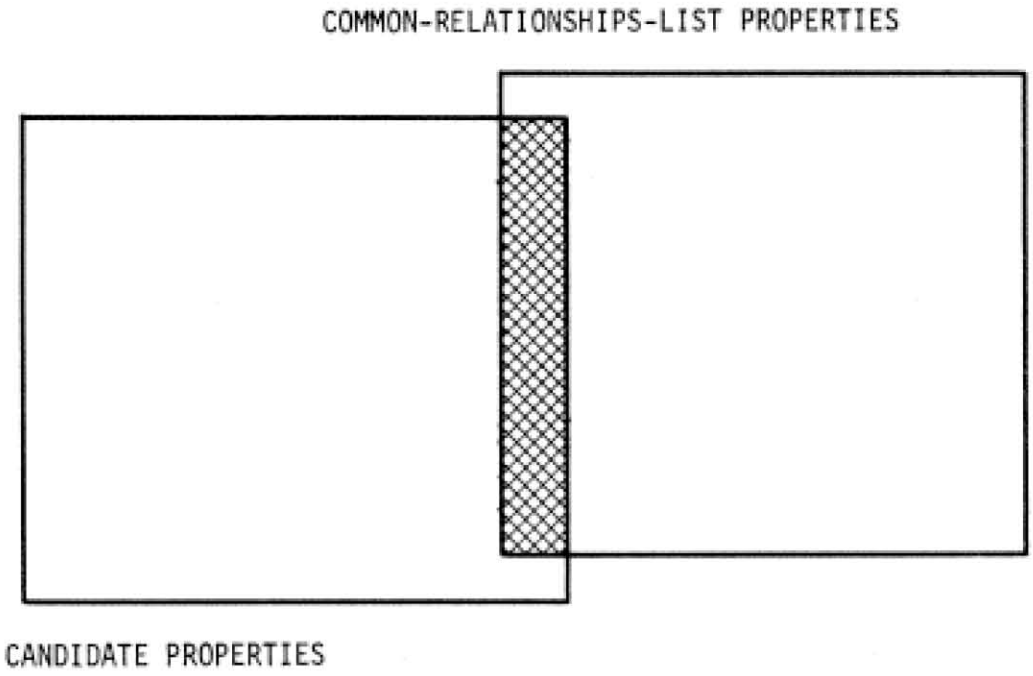


FIGURE 3-10

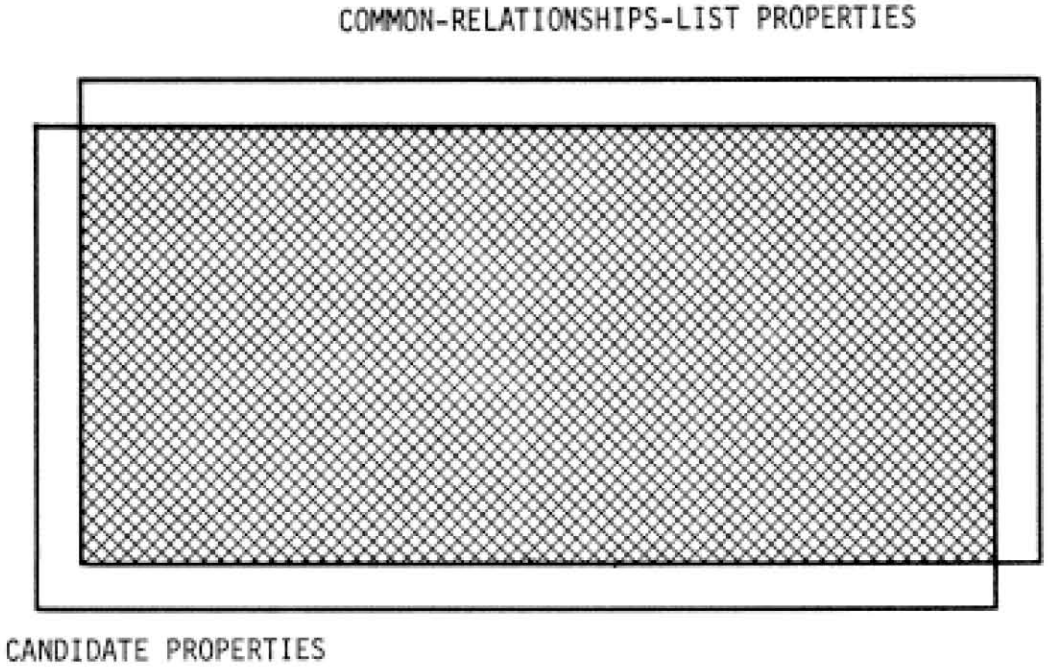


FIGURE 3-11

maximum possible.

Using this similarity formula to compare the various objects of the figure 3-9 example with the common-relationships-list, one has:

A versus the common-relationships-list --> 3/4 = .75

B versus the common-relationships-list --> 3/4 = .75

C versus the common-relationships-list --> 3/4 = .75

D versus the common-relationships-list --> 3/4 = .75

E versus the common-relationships-list --> 2/5 = .20

F versus the common-relationships-list --> 2/5 = .20

A, B, C, and D do not have scores of 1 only because the common-relationships-list does not yet have a property indicating size. The reason is that there is no size common to more than half of the currently possible group members, A, B, C, D, E, and F.

The much lower scores of E and F reflect the additional fact that as wedges they are different from the standard type. They are immediately eliminated according to the following general rule:

> Eliminate all candidate objects whose similarity scores are less than 80% of the best score any object attains.

This insures that the group will have members all with a nearly equal right to belong.

Next the process is repeated because those properties

common to the remaining candidates may differ from those properties common to the original group enough that one or more changes should be made to the common-relationships-list. This repetition continues until the elimination process fails to oust a candidate or fewer than three candidates remain.

After the first elimination of objects leaves A, B, C, and D, there is a new common-relationships-list:

common-relationships-list:

1 SUPPORTED-BY pointer to G
2 MARRYS pointer to G
3 A-KIND-OF pointer to BRICK
4 HAS-PROPERTY-OF pointer to MEDIUM-SIZE

Notice that there is now a size property since three of the four remaining objects have a pointer to medium size. The new comparison scores are:

A versus the common-relationships-list --> 4/4 = 1

B versus the common-relationships-list --> 4/4 = 1

C versus the common-relationships-list --> 4/4 = 1

D versus the common-relationships-list --> 3/5 = .6

This time D is rejected because its uncommon size causes a low score, leaving a stable group in which the objects are all quite like one another.

3.3 Other Kinds of Grouping

There obviously cannot be a single universal grooupinq procedure because attention must be paid not only to the scene involved, but also to the needs of the various programs that may request the grouping activity. I have discussed two grouping modes that programs can now do in response to various demands. There remain many others to be explored.

One of these involves looking for things that fit together. Children frequently do this at play without prompting, and adults do it extensively in solving jigsaw puzzles.

Another kind of grouping, one particularly sensitive to the goals of the request, is grouping on the basis of some specified property. The idea is to pick out all things satisfying some criteria, such as all the big standing bricks. The result could be a focusing of attention.

Still another way to group involves overall properties that are not obvious from purely local observations. Techniques here are again largely unexplored, but it seems that overall shape can sometimes impose unity on a complete hodge-podge. Figure 3-12 illustrates this point. All of the objects fit together to form a brick-shaped group. This is clearly not inherited from any consistency in how the parts are shaped or how they interact with their immediate

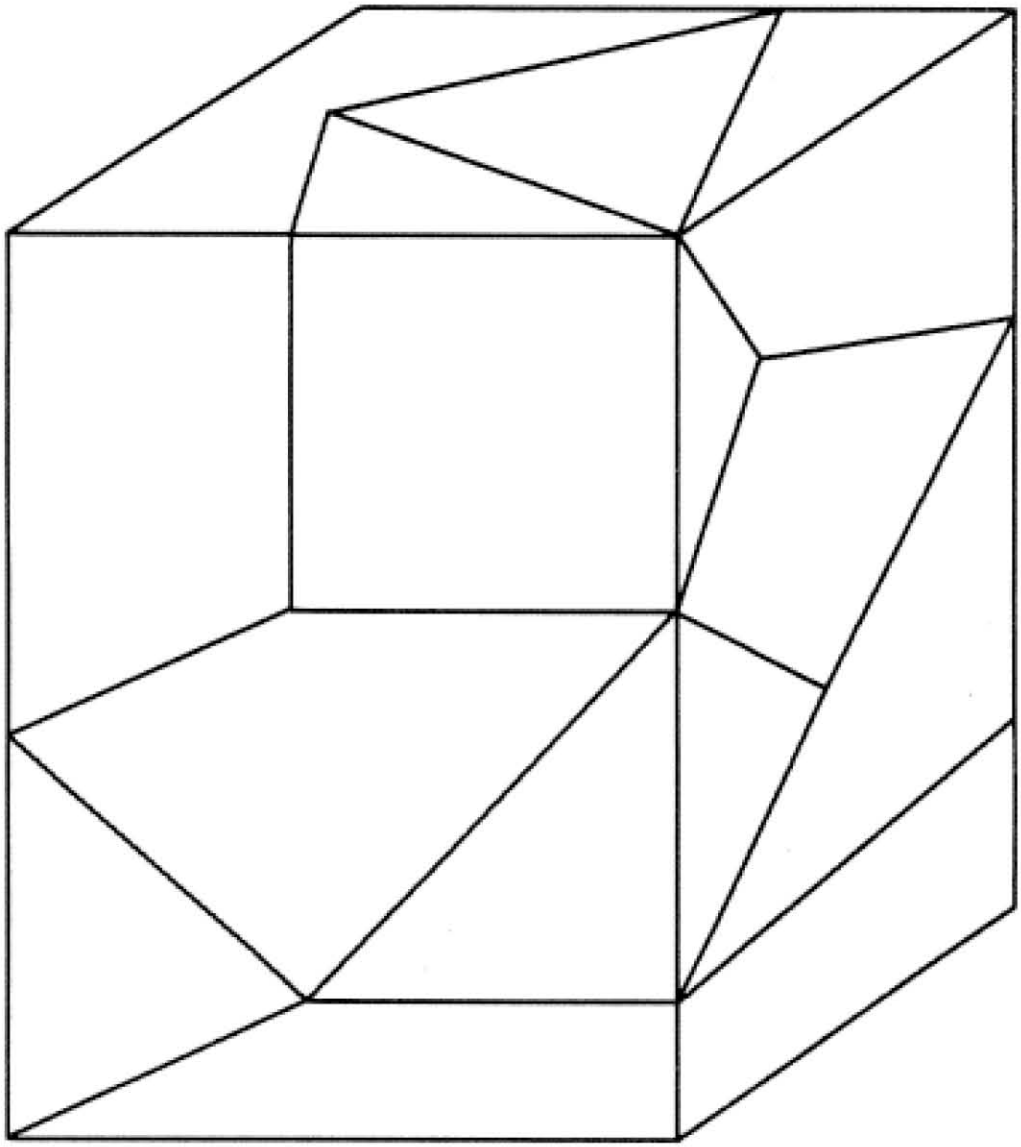


FIGURE 3-12

neighbors.

3.4 Describing a Group; The Typical Member

The machine needs some means of describing groups. The method it uses seems to work, but there is room for improvement.

First, the parts of the group are gathered together under a node created specifically to represent the group as a conceptual unit. Figure 3-13 illustrates this step for a group of three objects, A B and C.

Next comes a concise statement of what membership in the group means. This is done through the use of a typical-member node. Properties and relations that most of the group members share contribute to this node's description. If some group were composed of three standing bricks arranged in a tower, then the result would be the description shown in figure 3-14. The typical member is there described as a kind of brick, as standing, and as on top of another member of the group. Notice also the FORM pointer to SEQUENCE which indicates the kind of group formed.

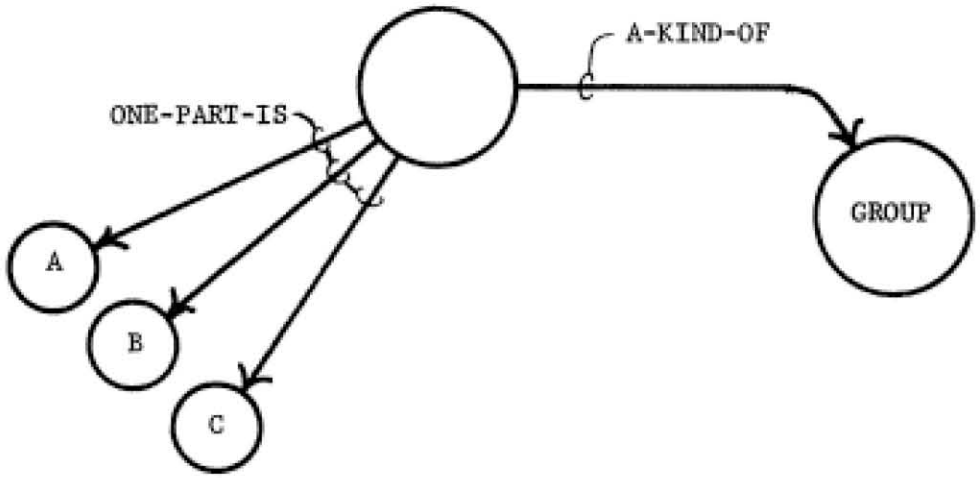


FIGURE 3-13

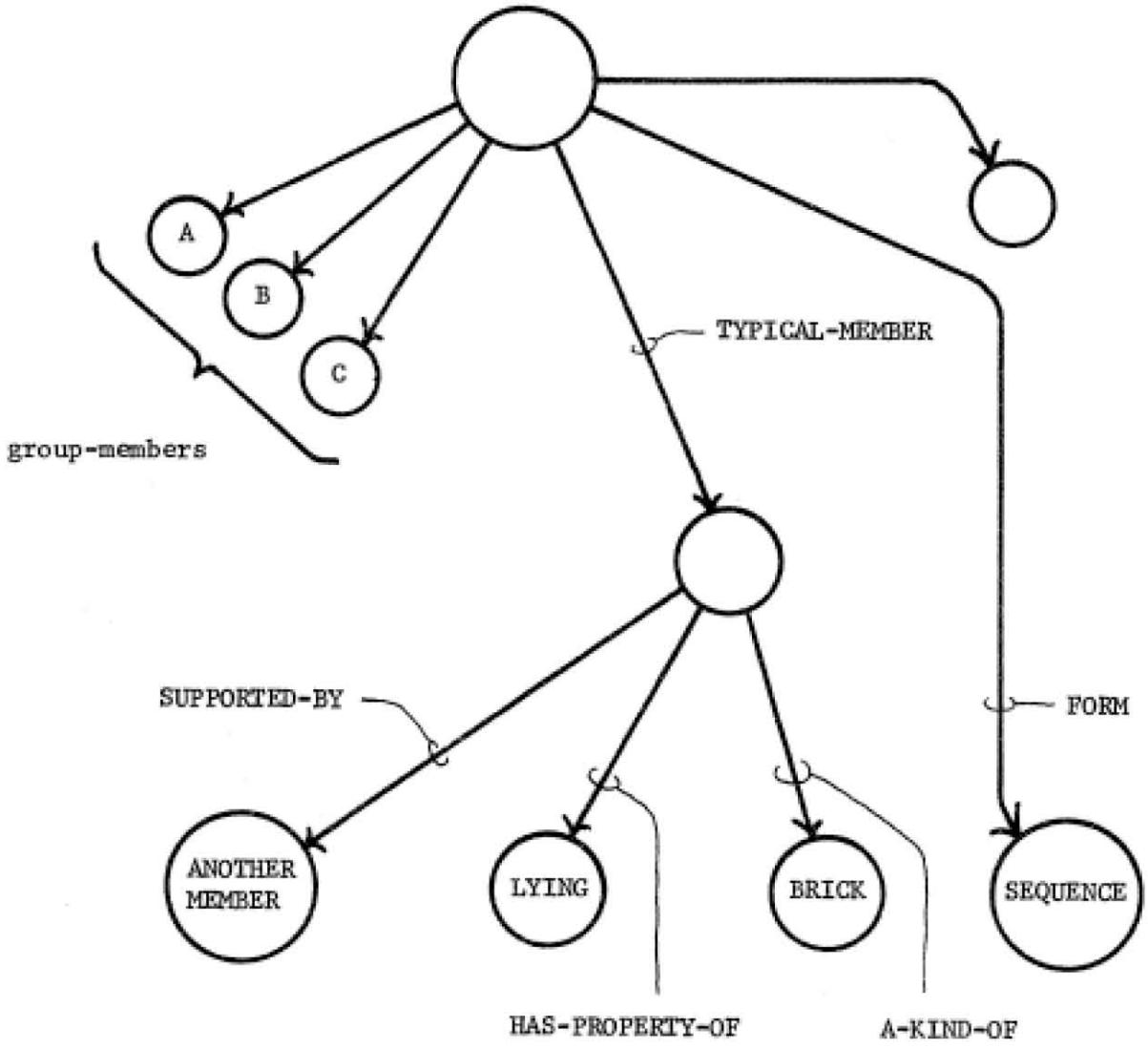

A
B
C
group-members
TYPICAL-MEMBER
SUPPORTED-BY
FORM
ANOTHER MEMBER
LYING
BRICK
SEQUENCE
HAS-PROPERTY-OF
A-KIND-OF

FIGURE 3-14

4 Similarities and Differences

4.1 Network Matching

Powerful scene description programs are essential to scene comparison and identification. Matching is equally important since the machine must know which parts of two descriptions correspond before it can compute similarities and differences. Figure 4-1 briefly illustrates. A process explores the two descriptive networks and decides which nodes of the two best correspond in the sense that they have the same function in their respective networks. The nodes in a pair that so correspond are said to be linked to each other. The job of the matching program is simply to find the linked pairs. Node LC and node RC in figure 4-1 both have only A-KIND-OF pointers to BRICK. Since no other nodes have similar descriptions, it is clear that LC and RC should be a linked pair. Similarly, LB and RB should be a linked pair since both have A-KIND-OF pointers to WEDGE and both have SUPPORTED-BY pointers to parts of a pair of nodes already known to be linked.

Of course the job of the matching program is not so easy when the two scenes and the resulting two networks are not identical. In this case the process forms linked pairs involving nodes that may not have identical descriptions, but seem most similar nevertheless. More details are described

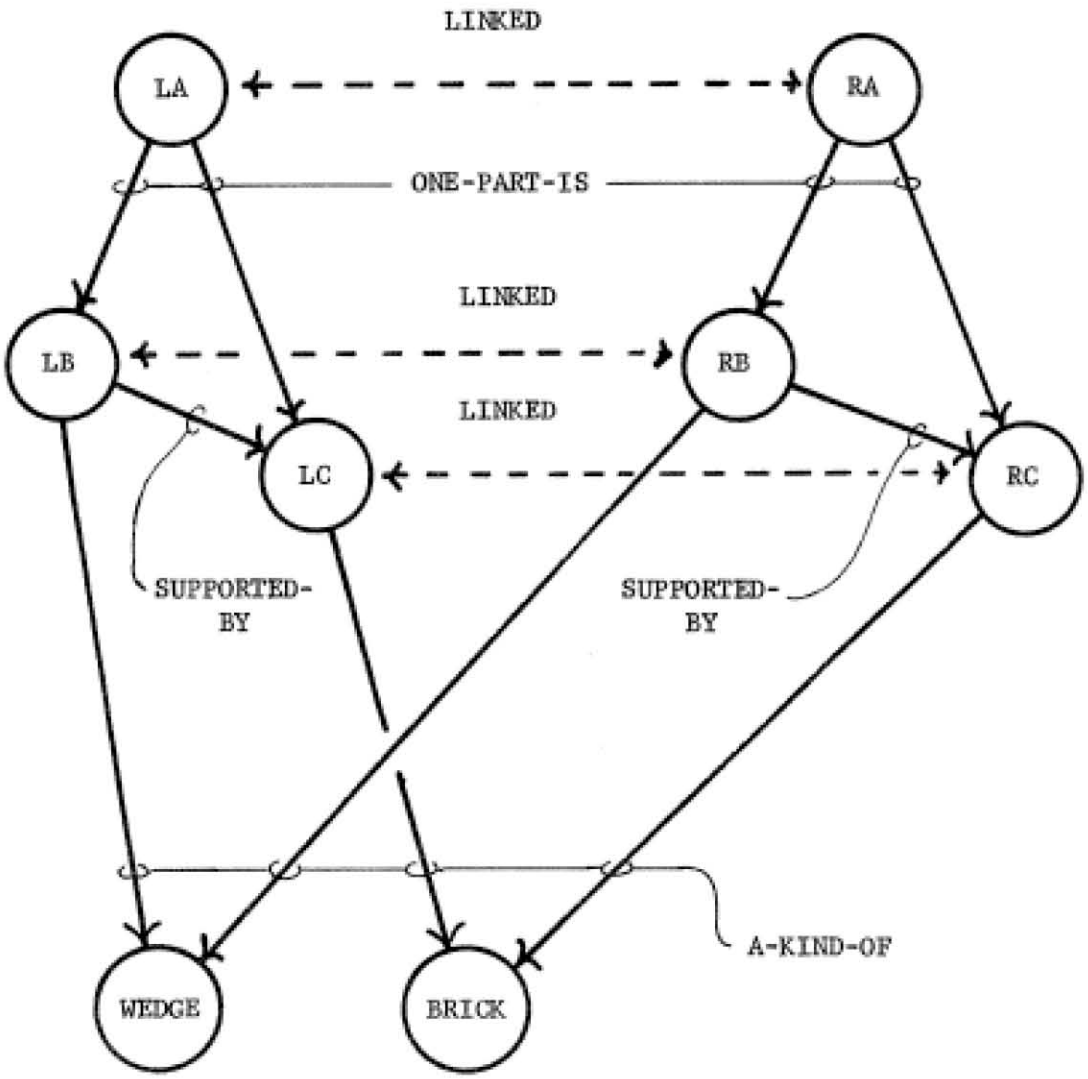

LINKED
LA
RA
ONE-PART-IS
LINKED
LB
RB
LINKED
LC
RC
SUPPORTED-
BY
SUPPORTED-
BY
A-KIND-OF
WEDGE
BRICK

FIGURE 4-1

in the appendix.

4.2 The Skeleton

Once the matching process has examined two networks and has established the linked pairs of nodes, then description of network similarities proceeds. The result is simply a new chunk of network that describes those parts of the compared networks that correspond. This chunk is called the skeleton because it is a framework for the rest of the comparison description. As figure 4-2 suggests, each linked pair contributes a node to the skeleton. Certain pointers connect the new nodes together. These occur precisely where the compared networks both have the same pointer from one member of some linked pair to a member of some other linked pair. Notice that the skeleton is basically a copy of the structure that the compared networks duplicate.

4.3 Comparison Notes

Complete comparison descriptions consist of the skeleton together with a second group of nodes attached to the skeleton like grapes on a grape cluster. Each of the nodes in this second category is called a c-note, short for comparison note. The most common type of c-note is the intersection c-note which describes the situation in which both members of a linked pair point to the same concept with the same pointer. Suppose, for example, that a pair of

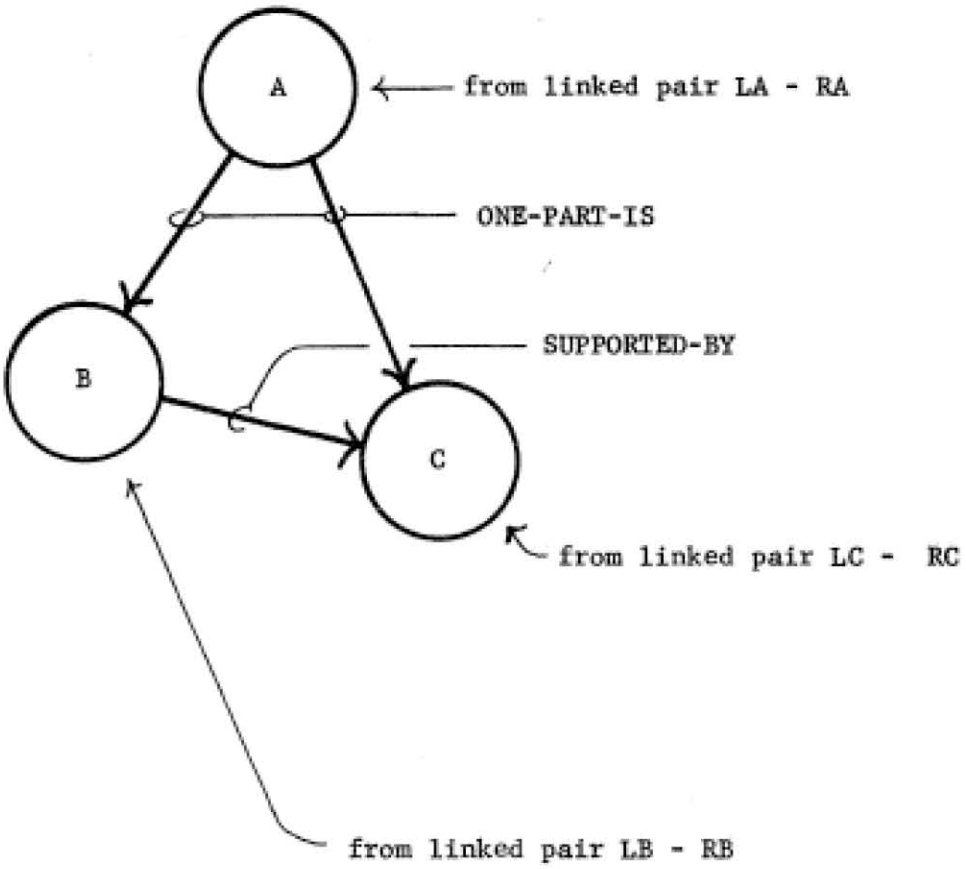

A
from linked pair LA - RA
ONE-PART-IS
B
SUPPORTED-BY
C
from linked pair LC - RC
from linked pair LB - RB

FIGURE 4-2

corresponding objects from two scenes are both wedges. Then both concepts exhibit an A-KIND-OF pointer to the concept WEDGE. (figure 4-3) In English one can say:

1. There is something to be said about a certain linked pair.
2. There is an intersection involved.
3. The associated pointer is A-KIND-OF.
4. The intersection occurs at the concept WEDGE.

Figure 4-4 shows how each of these simple facts translates to a network entry. First, a pointer named C-NOTE extends from the skeleton concept corresponding to the linked pair to a new concept that anchors the intersection description. The A-KIND-OF pointer identifies this concept as a kind of intersection. Finally other pointers identify the pointer, A-KIND-OF, and the concept, WEDGE, associated with the intersection.

All of the c-notes look like this intersection paradigm.

.3.1 Digression: Evans' Program

Embodying difference descriptions in the same network format permits operation on those descriptions with the same network programs. Thus two difference descriptions can be compared as handily as any other pair of descriptions.

Those familiar with Tom Evans' vanguard program, ANALOGY [4], can understand why this is a powerful feature, rather

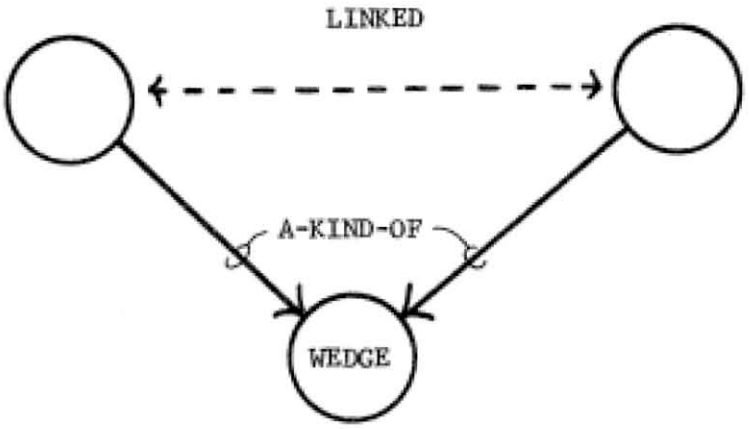


FIGURE 4-3

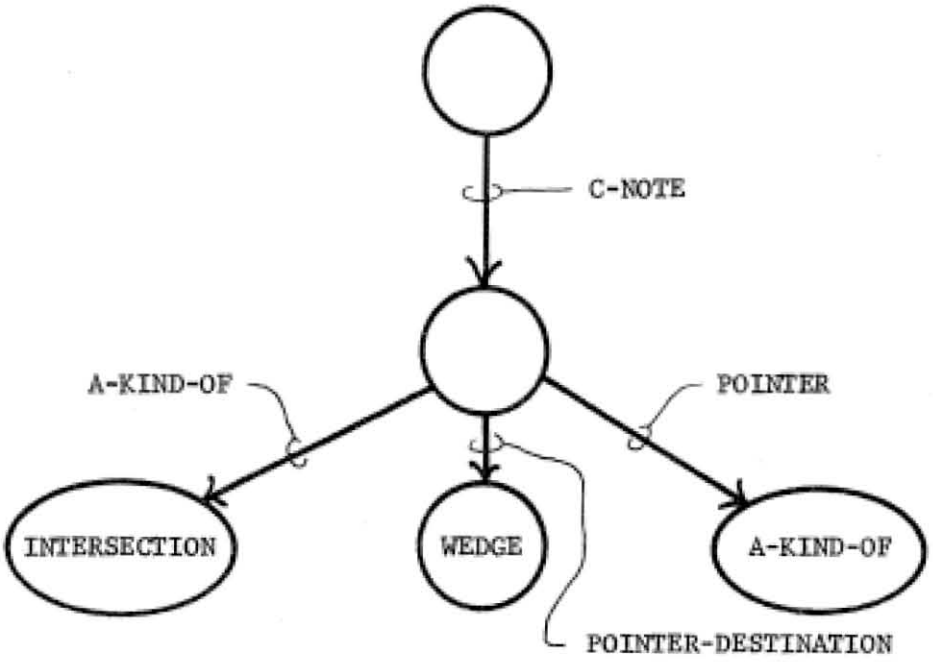


FIGURE 4-4

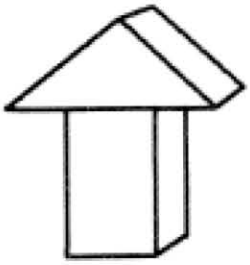
than simply a contribution toward memory homogeneity. Evans' program worked on two dimensional geometric figures rather than drawings of three dimensional configurations. Nevertheless his ideas generalize easily and fit nicely into the vocabulary used here.

Figure 4-5 suggests the standard sort of intelligence test problem involved. The machine must select the scene X which best completes the statement: A is to B as C is to X. In human terms one must discover how B relates to A and find an X that relates to C in the same way.

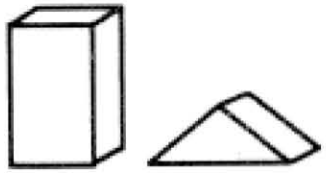
Using the terminology of nets and descriptions, one solution process can be formalized in the following way: First compare A with B and denote the resulting comparison-describing network by

$$d[A:B].$$

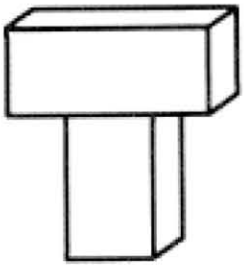
Similarly compare C with the answer figures generating descriptions of the form d[C:X]. The result is a complete set of comparisons describing the transformations that carry one figure into another. Next one should compare the description of the transformation from A to B, d[A:B], with the others to see which is most like it. The best match is associated with the best answer to the problem. If M is a metric on comparison networks that measures the difference between the compared networks, one can say



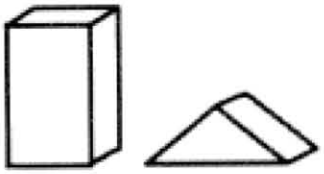
A



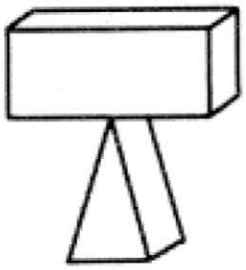
B



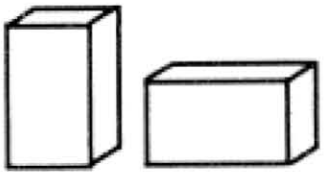
C



ONE

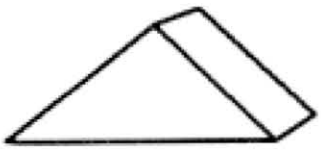


TWO



THREE

FOUR



FIVE

FIGURE 4-5

choose X such that

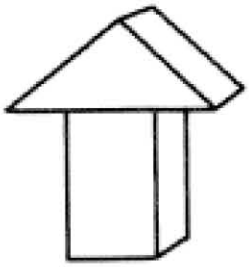
M(d[d[A:B]:d[C:X]])

is minimum

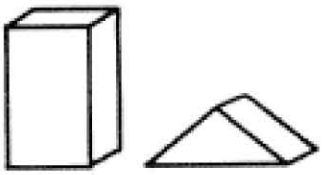
The metric I use is not fancy. It is the one discussed later in chapter 7 that serves to identify some scene with some member of a group of models. It works because that problem entirely parallels the problem of identifying a given transformation description with some member of a group. The identification program, together with a short executive routine, handles the problem of figure 4-5 easily, correctly reporting scene three as the best answer. Reasonably enough, the machine thinks scene one is the second best answer.

The machine does as well on the slightly altered problem in figure 4-6, reporting four as the best answer.

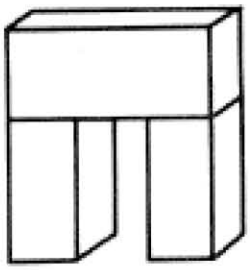
Of course if the machine's answers are to be those of the problem's formulator, then the machine's describing, comparing, and comparison measuring processes should all give results that resemble his. Moreover, a really good analogy program should have available alternatives to its basic describing, comparing, and comparison measuring processes. Then in the event no single answer is much better than the others, the program can try some of its alternatives as one or more of its basic functions must not be operating according to what the problem maker intended. Evans' program



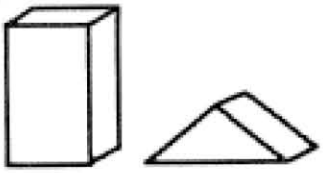
A



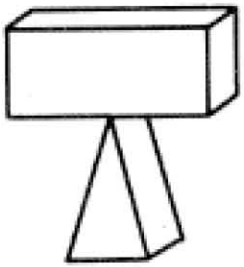
B



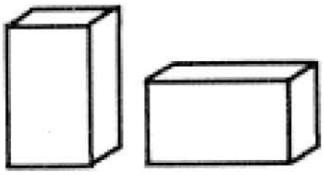
C



ONE



TWO



THREE

FOUR

FIVE

FIGURE 4-6

is superior to mine in this respect because it can often compare two drawings in more than one way. It can visualize the change in figure 4-7, for example, as either a reflection or any one of several rotations.

Given my formulation of the analogy problem, it is easy to see how certain interesting generalizations can be made. After all, once an X is selected, the network symbolized by d[d[A:B]:d[C:X]] describes the problem, and as a description, it can be compared with the descriptions of other problems. By thus applying the comparison programs for the third time, one can deal with the question, Analogy problem alpha is most like which other analogy problem? Alternatively, one can apply the analogy solving program to problem descriptions instead of scenes and answer the question, Analogy problem alpha is to analogy problem beta as analogy problem gamma is to which other analogy problem? This involves four levels of comparison. But of course there is no limit, and with time and memory machines could happily think about extended analogy problems involving an arbitrary number of comparison levels.

4.3.2 Another Digression: Newell, Shaw, and Simon's Program

A classic piece of work in artificial intelligence is that of Newell, Shaw, and Simon on the scheme known as the General Problem Solver, always abreviated GPS [5]. One form

FIGURE 4-7

of GPS provides another example of how comparisons may be usefully compared with other comparisons.

From an abstract point of view, GPS involves the notion that problems may be thought of in terms of some solution or goal, G, together with a current state C. Additonally there are operators, O(i), that convert classes of states into others. One may abreviate their action by writing

O(i):F-IN(i) --> F-OUT(i),

meaning that operator O(i) tends to convert states of the form F-IN(i) into states of the form F-OUT(i).

GPS notices the difference between the current state C and the desired state G and then tries to apply an operator relevant to reducing that difference. This produces a new current state somewhat closer to the desired state. Applying this process iteratively, GPS may eliminate the difference between C and B, thereby solving the problem.

In early versions of GPS the programmers supplied a table giving the relevant operations for all the differences between C and G that might be observed. But later on Newell described an approach [6] that I think may be more transparently represented using the same notions of second order comparison minimization that is useful in discussing analogy problems. The idea is that the operators, O(i), may be described by the difference between their input and output

forms,

d[F-IN(i):F-OUT(i)].

Then Newell feels it is heuristically sound to apply the operator whose description is most like the difference between the current and desired states,

d[C:G].

One can say more formally,

choose the operator O(i) such that

M(d[d[F-IN(i):F-OUT(i)]:d[C:G]])

is minimum.

Notice that the selection of an operator is curiously like solving an analogy problem for which one chooses a pair, (X(i),Y(i)) from a set of offered pairs that best completes the statement: A is to B as X(i) is to Y(i).

4.4 A Catalogue of C-note Types

4.4.1 The Supplementary-pointer

Consider the scenes in figure 4-8 and their descriptions in figure 4-9. Scene L has the pointer SUPPORTED-BY between LA and LB, but scene R does not have a pointer between the objects linked to LA and LB. The note describing this situation is called a supplementary-pointer c-note and has the form shown in figure 4-10.

Figure 4-11 suggests a related situation. Here the linked concepts L and R differ only in that L has an

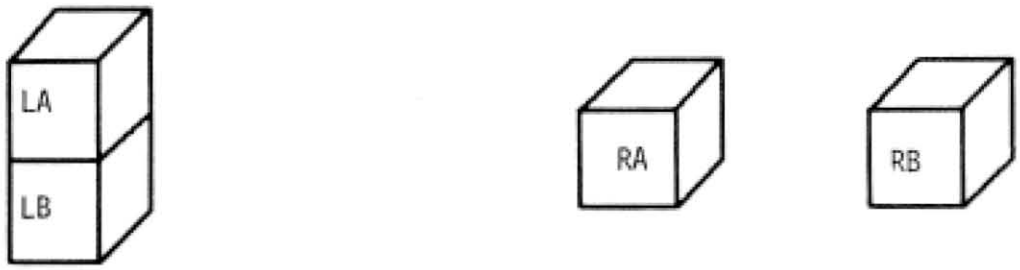


FIGURE 4-8

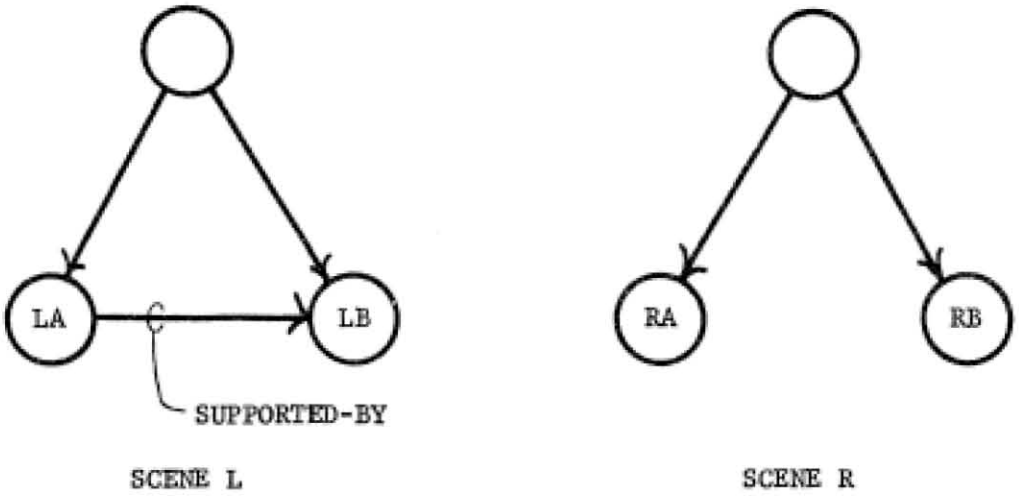


FIGURE 4-9

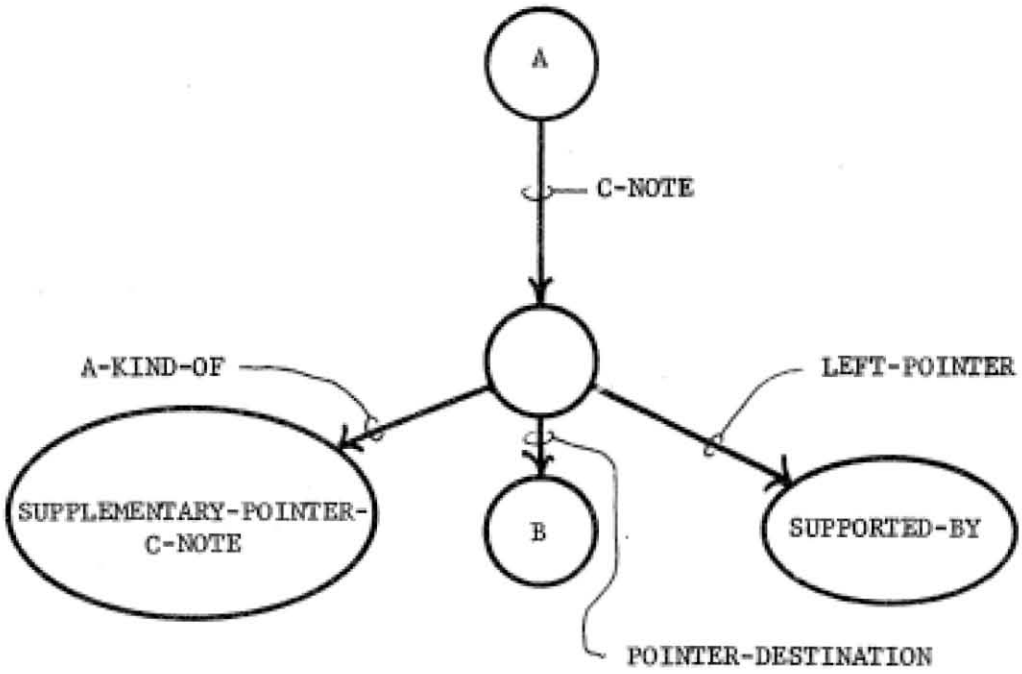


FIGURE 4-10

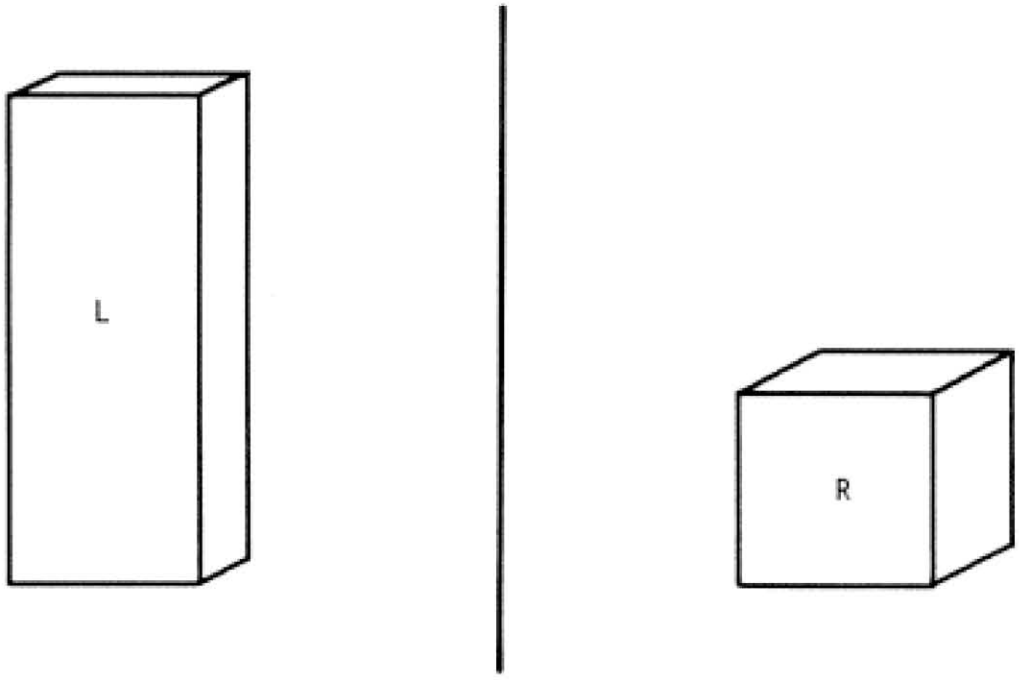


FIGURE 4-11

additional pointer identifying it as standing. This differs from the supplementary-pointer case in that STANDING is not linked to anything in the other scene. A pointer to the concept EXIT signals this situation. (figure 4-12) Exits involve concepts generated by the scene description program as well as concepts like STANDING that reside in the net permanently. If one scene contains more objects than another, the concepts left over and not matched end up in exit packages.

4.4.2 Pointer Modifications

Suppose the networks in figure 4-13 are compared. Notice the MARRYS pointer between LA and LB and the DOES-NOT-MARRY pointer between RA and RB. These could be handled individually as unrelated supplementary-pointer c-notes, but this would ignore the close relationship between MARRYS and DOES-NOT-MARRY. Consequently a different type of c-note is used that recognizes the relationship. It is the negative-satellite-pair c-note. With it, the comparison looks as shown in figure 4-14. To find such negative-satellite-pair c-notes, the comparison programs peruse the descriptions of unmatched pointers between linked pairs for evidence of relationship. For example, MARRYS is described in part by a NEGATIVE-SATELLITE pointer to DOES-NOT-MARRY. Now of course there are other pointers that are also just one step removed

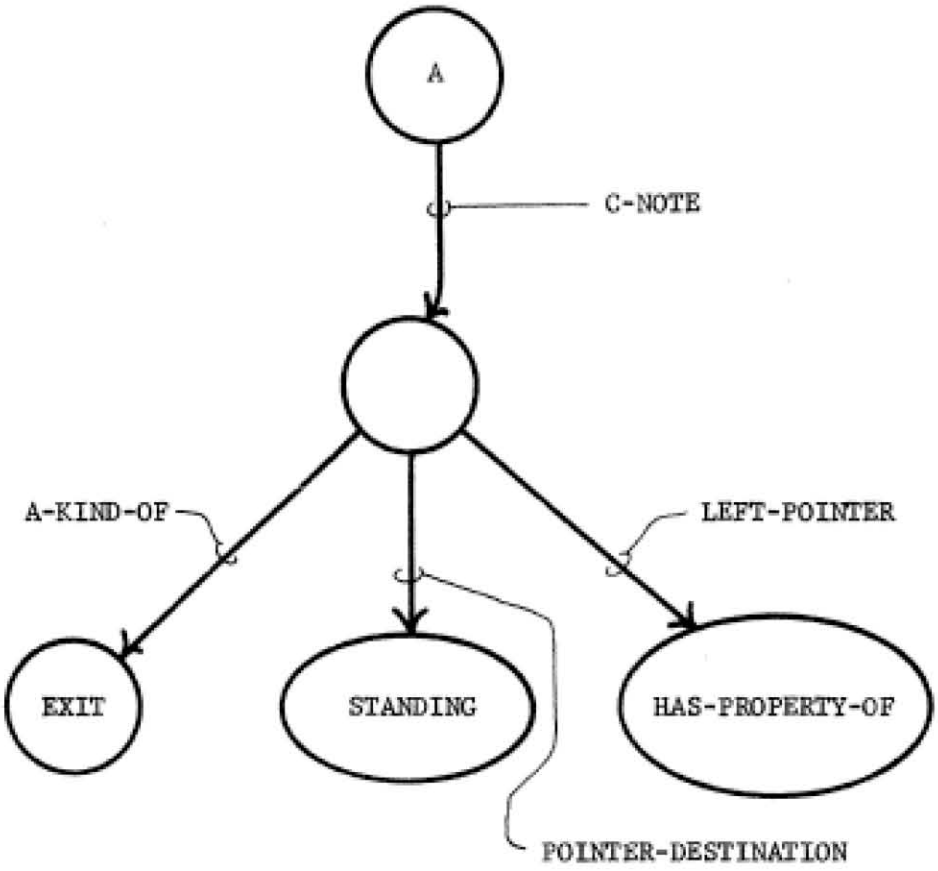

A
C-NOTE
A-KIND-OF
LEFT-POINTER
EXIT
STANDING
HAS-PROPERTY-OF
POINTER-DESTINATION

FIGURE 4-12

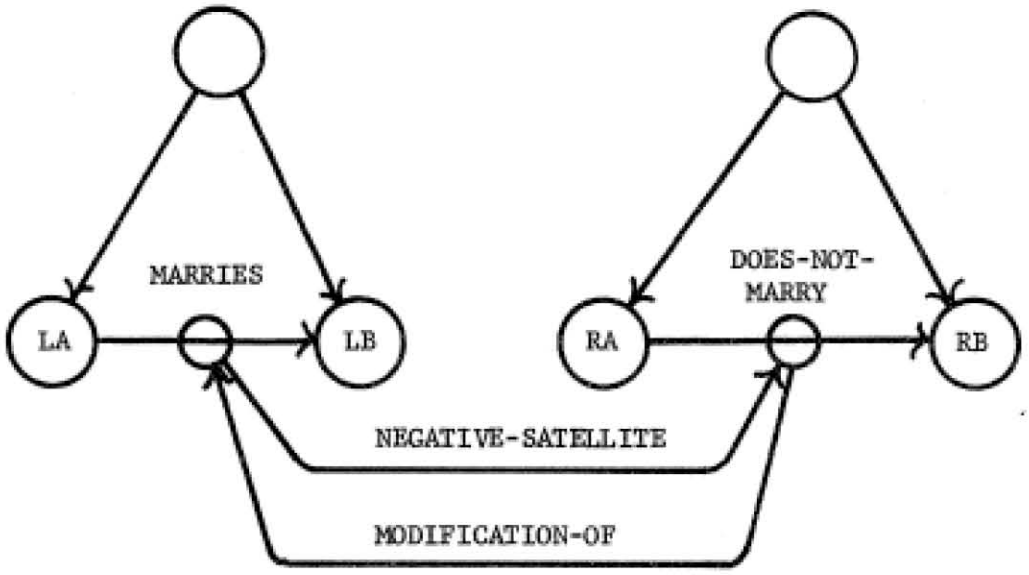


FIGURE 4-13

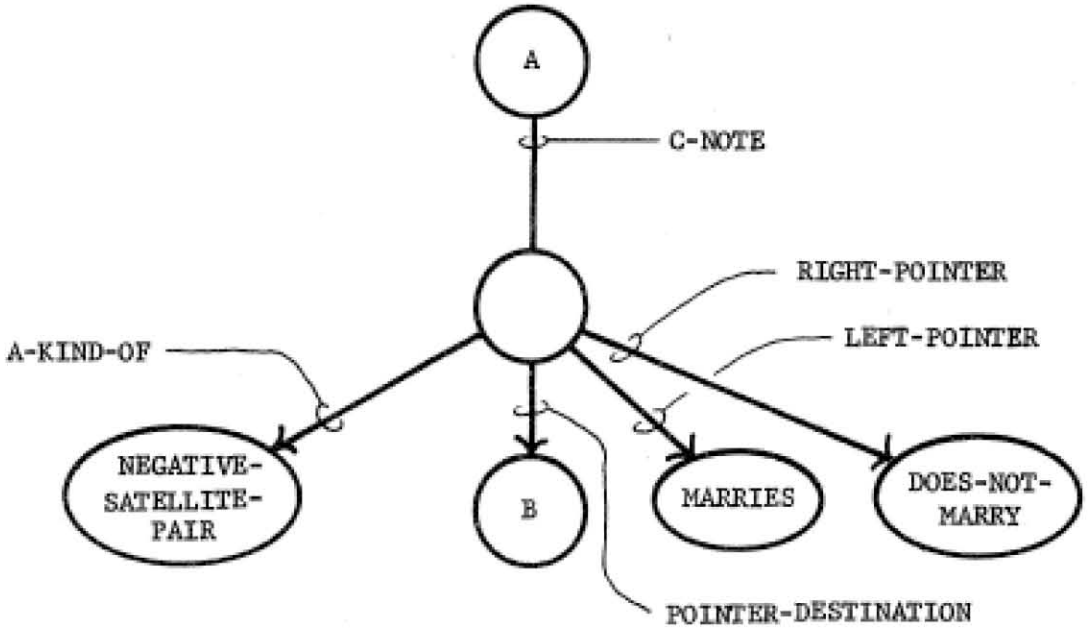


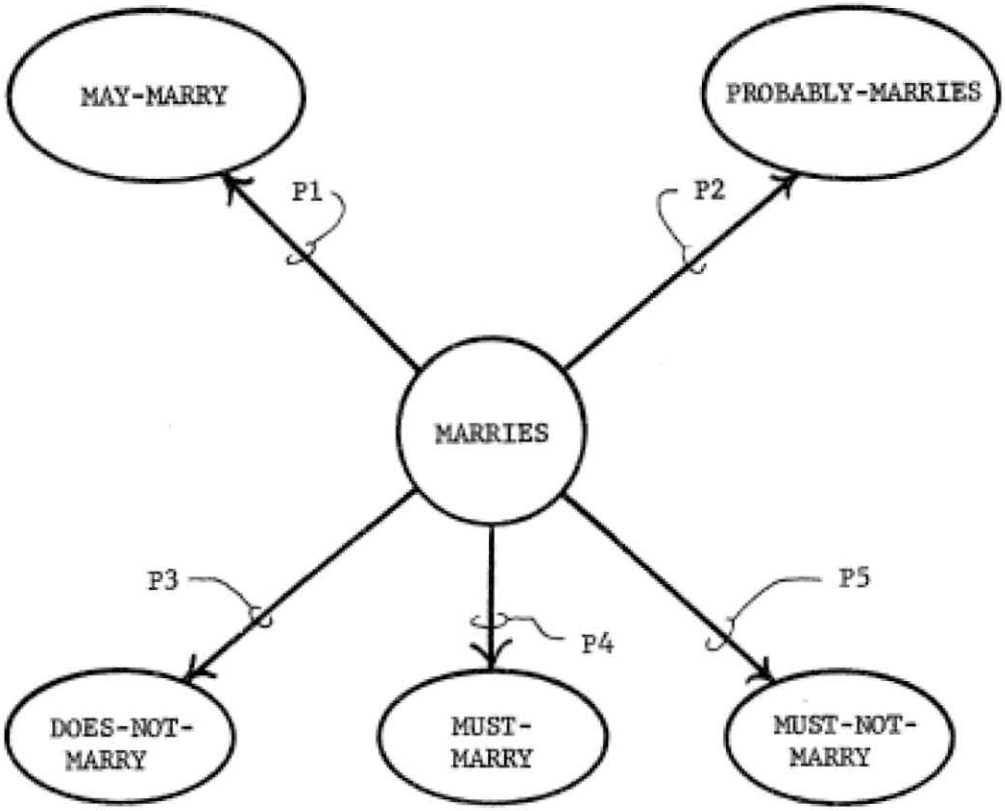
FIGURE 4-14

from a basic relation. All such pointers that are modifications of the basic relation are called satellites because they cluster around the basic relation to which they are attached by the pointer MODIFICATION-OF. Uncertainty, for example, is expressed by PROBABLY satellites or MAYBE satellites. The MUST satellites and the MUST-NOT satellites are others of particular importance in model construction. These inform the model matching programs that the presence or absence of some pointer is vital if some unidentified network is to be associated with a particular model network containing such a pointer. Figure 4-15 shows some of the satellites of MARRYS.

Each type of satellite is associated with a type of c-note forming an open ended family. Thus in addition to negative-satellite-pair c-notes, there are probably-satellite-pair c-notes, maybe-satellite-pair c-notes, must-satellite-pair c-notes, must-not-satellite-pair c-notes and so on.

4.4.3 Concept Modifications

Frequently the members of a linked pair both have pointers to closely related concepts. For example, if a brick in one scene is linked to a cube in another, the situation is as shown in figure 4-16. This is very much like the pointer-satellite idea with A-KIND-OF replacing



P1 = MAYBE-SATELLITE
P2 = PROBABLY-SATELLITE
P3 = NEGATIVE-SATELLITE
P4 = MUST-BE-SATELLITE
P5 = MUST-NOT-BE-SATELLITE

FIGURE 4-15

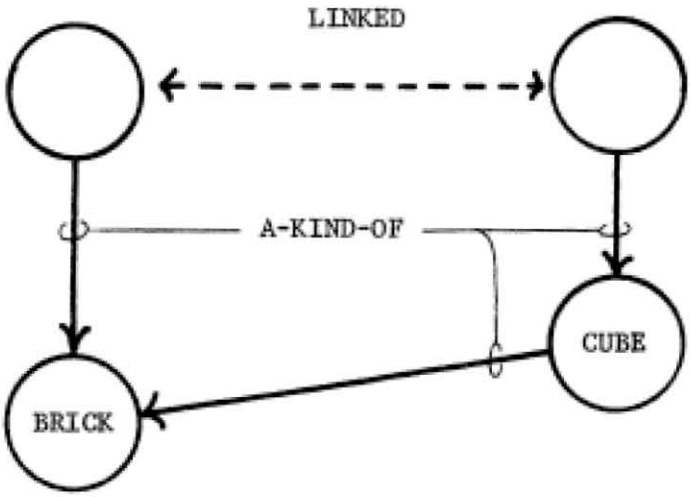


FIGURE 4-16

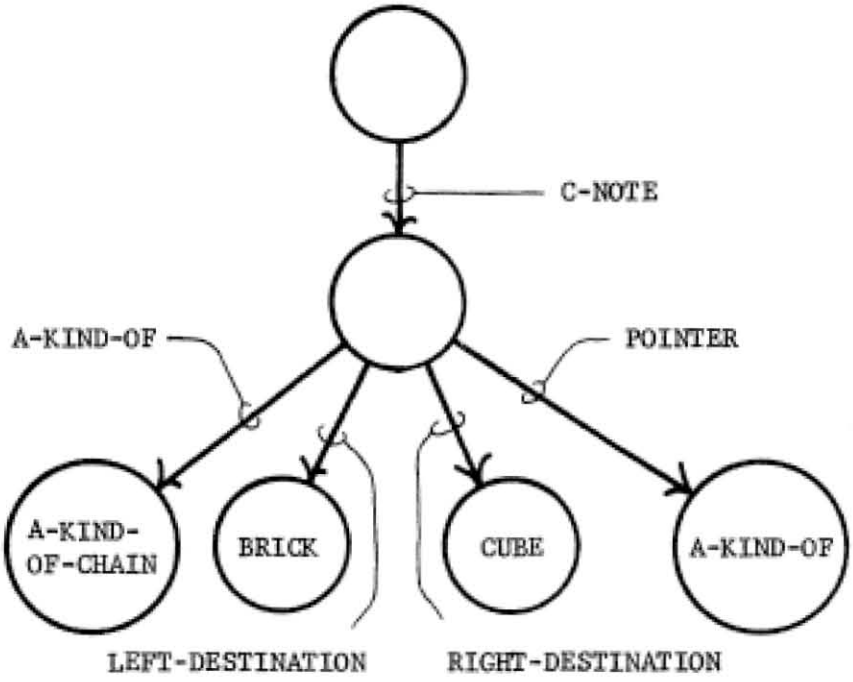


FIGURE 4-17

MODIFICATION-OF. In any case, the description generator recognizes this and similar situations and again generates a group of c-note types. The first of these is the A-KIND-OF chain illustrated by the above situation. This causes the c-note of figure 4-17.

The a-kind-of-chain c-note also includes situations in which one concept is related to another not directly, but rather through two or three A-KIND-OF relations. Suppose, for example, a cube is linked with an object for which no identification can be made. There is still an a-kind-of-chain c-note because cube is linked to the general concept, OBJECT, by a sequence of A-KIND-OF relations. (figure 4-18)

Another kind of popular concept modification is the a-kind-of-merge c-note. These a-kind-of-merge c-notes occur if there is no A-KIND-OF chain as described above, but each concept has a chain of A-KIND-OF pointers to some third concept. For example, WEDGE and BRICK are both connected to the concept, OBJECT, by A-KIND-OF. (figure 4-19)

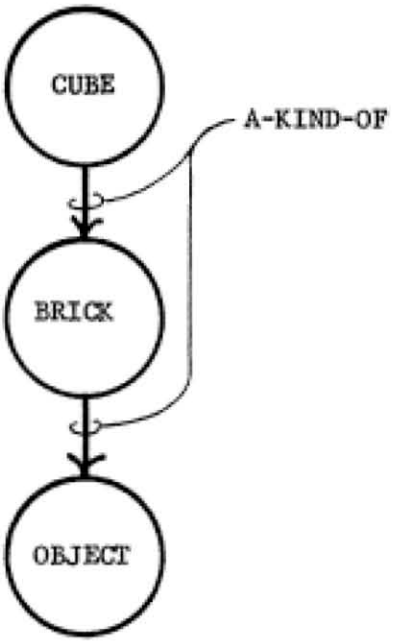


FIGURE 4-18

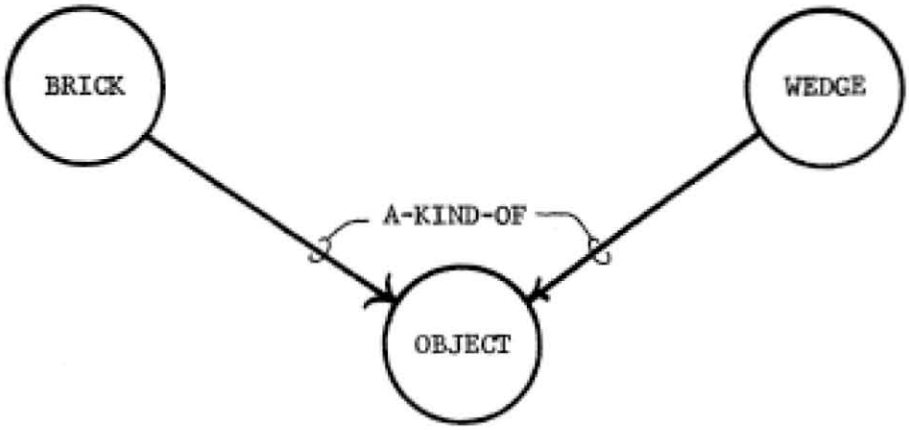


FIGURE 4-19

5 Learning and Model Building

5.1 Learning

In this chapter I discuss learning to recognize simple block configurations. Although this may seem like a very special kind of learning, I think the implications are far-ranging.

It is possible to assuime extreme positions on the subject of learning. One person may think learning to do things is very complicated, while learning to recognize things is comparatively simple, because one merely aquires templates or some such. Conversely, another person may think learning to do is simple, but learning to recognize involves deep Gestaltist problems of synthesis or other impedimenta.

My opinion is that learning by examples, learning by being told, learning by imitation, learning by reinforcement and other forms are much like one another.

In the literature there is frequently an unstated assumtion that these various forms are fundamentally different. But I think the classical boundaries between the various kinds of learning will disappear, once superficially different kinds of learning are understood in terms of processes that construct and manipulate descriptions. No kind of learning need be desperately complicated once the descriptive machinery is available, but all constitute

opaque, intractable processes without it.

Then once the problem of using descriptions is thoroughly understood, it will be possible to give meaningful thought to a deeper problem, that of learning to use descriptions. It does not seem simple, but using this point of view, it seems less than impossible.

> The notions of learning and teaching are broad and confused. Generally, people think of these things as occurring together, so that whenever something learns, something else teaches. But somehow intuitive notions fail when it comes to thinking about machines. Computers can now play tolerable and improving chess and do marvelous symbolic integrations. Yet while people freely use the word "teach" in describing what the programmers do, hardly anyone thinks of the machine as learning.
>
> The reason seems to be that the human programmer has supplied so much detail that the machine is more a mimic than a thing with learning ability. The machine acquires its skill without ever really knowing what is going on or how it might improve without the laborious services of an information processing surgeon. The unfortunate machine is in the position of school pupils who know facts and perhaps memorize simple algorithms, but are never programmed to learn. The programs to be discussed here are an effort to show that a machine can do better and can learn in a realistic sense, given a chance.

5.2 Descriptions and Models

I want to make a clear distinction between a description of a particular scene and a model of a concept. A model is like an ordinary description in that it carries information about the various parts of a configuration. But a model is more in that it exhibits and indicates those relations and

properties that must and must not be in evidence in any example of the concept involved.

Suppose, for example, the description generating programs report the following facts in connection with the arch in figure 5-1:

1. Object A is a brick.
2. Object A is supported by B and C.

Now suppose the description containing these facts were compared with the scene in figure 5-2, where object A is a wedge, and with the scene in figure 5-3, where object A lies on the table. In both cases comparison could be made and differences appropriately noted, but the identification of one or the other of these new scenes as arches would be risky indeed because so far the machine knows only what one arch looks like without knowing what in that description is important!

Humans, however, have no trouble identifying the scene in figure 5-2 as an arch because they know that the exact shape of the top object in an arch is unimportant. On the other hand, no one fails to reject the scene in figure 5-3 because the support relations of the arch are crucial. Consequently, it seems that a description must indicate which relations are mandatory and which are inconsequential before that description qualifies as a model. This does not require

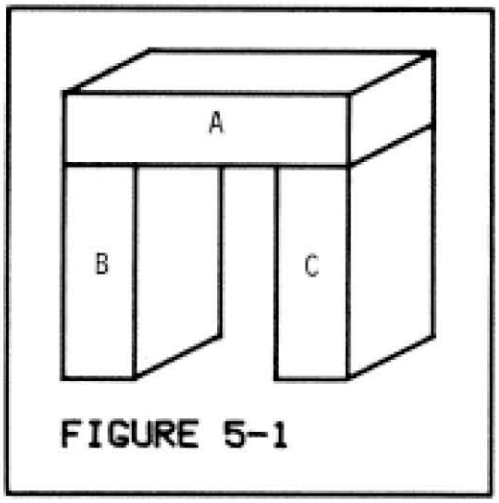


FIGURE 5-1

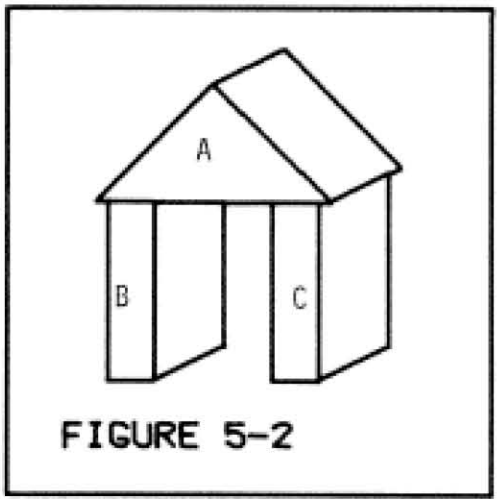


FIGURE 5-2

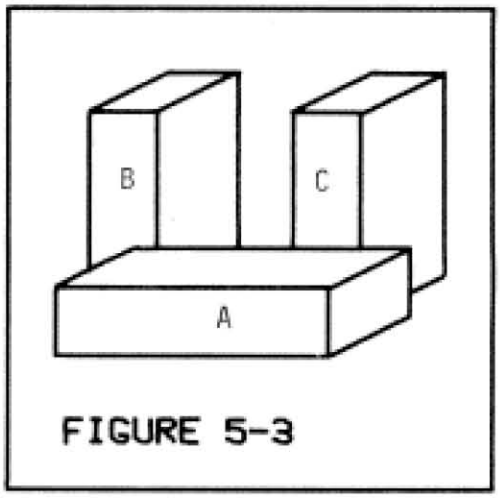


FIGURE 5-3

any descriptive apparatus not already on hand. One need only substitute emphatic forms like MUST-BE-SUPPORTED-BY for basic pointers like SUPPORTED-BY or, in some cases, add new pointers. Discovering where and when to perform these operations can be somewhat involved, however, and requires the bulk of this chapter for discussion.

5.3 Examples, Near-misses, and Non-examples

Suppose it is desirable to train a machine to recognize the letter A without restriction as to type size or font. The designer then has two sets of options: He must decide how his machine is to work; and he must decide what to show the machine. One idea is to show the machine vast numbers of As and hope that it will benefit from such an experience by somehow noticing the features which appear repeatedly. But this assumes that the frequently seen properties are essential ones, which can be a bad rule. Moreover, there is little possibility for skillful teaching. There is no obvious way the teacher could quickly convey a particular idea such as the notion that the crossbar of an A is important, even if the teacher realized it. Finally, such samples generally only suggest properties a candidate for match should have -- it is hard for them to indicate forbidden properties.

This is true because expert description programs

> would be needlessly overburdened and would promote confusion if they were always to indicate all negative properties and those not observed. Therefore a property's consistent absence can be easily overlooked. If the description program does list all properties in spite of inefficiency, then many properties are statistically likely to be missing from a short training sequence. But the statistic-gathering machinery would think such properties should not be present in examples of the model, even though their absence was by coincidence.

One might attempt to train basically the same machine to handle a whole repertoire of concepts by showing examples of each member. Thus if the repertoire were the alphabet, examples of all the letters would be shown, rather than just As. This shifts the question to Which description does an unknown best match? It still avoids the more fundamental question, What is it about each character that is essential and permits it to be recognized? The machine now may use As and non-As, but the difficulties are only obscured, not circumvented. There remains no way to directly convey an idea, and there remains the fallacy that frequent appearance means importance. The problem of indicating what properties preclude identification with a particular model is only tangentially and occasionally addressed in that sometimes a property converts one description into another, as in the case of adding a forbidden midline crossbar to a C. The machine does not know a C cannot have a crossbar; it only knows such a crossbar makes a figure more like an E.

In my judgement, near misses are the really important examples in learning. In conveying the idea of an arch, an arch certainly should be shown first. But then there should be some samples that are not arches, but do not miss being arches by much. Small differences permit the machine to localize some part of its current opinion about a concept for improvement. If one wants the machine to learn that the uprights of an arch cannot marry, one should show it a scene that fails to be an arch only in this respect. If the machine is to know a C cannot have a crossbar, it should see a character that fails to be a C only because of a crossbar. Such carefully selected near misses can suggest to the machine the important qualities of a concept, can indicate what properties are never found, and permit the teacher to convey particular ideas quite directly.

> It is curious how little there is in the literature of machine learning about mechanisms that depend on good training sequences. This may be partly because previous schemes have been too inadequete to bear or even invite extensive exploration of this centrally important topic. Perhaps there is also a feeling that creating a training sequence is too much like direct programming of the machine to involve real learning. This is probably an exaggerated fear. I agree with those who believe that the learning of children is better described by theories using the notions of programming and self-programming, rather than by theories advocating the idea of self-organization. It is doubtful, for example, that a child could develop much intelligence without the programming implicit in his instruction, guidance, closely supervised activity, and general interaction with other humans.

5.4 Model Development

The machine's model building program starts with a description of some example of the concept to be learned. This description is itself the first model of the concept. Subsequent samples are either examples of the concept or near misses. These examples and near misses reveal weaknesses and lead to a new models. Section 5.5 shows in some detail how the comparison between the current model and the description of a new sample produce a new model in the case where only one difference is found.

One then has a sequence of more and more sophisticated models. See figure 5-4. Frequently, several responses may appropriately address the comparison between the current model and a new sample. When this happens, branches occur in the model development sequence and it is convenient to talk about a tree of models. Figure 5-5 shows such a tree. Section 5.6 discusses how the alternative branches come about. The machine selects one branch at each point for further development. The meandering path leading from the top of the tree down to the current model is called the main line. The main line changes course when a particular sequence of branch selections leads to untenable situations. Section 5.7 describes how and when this happens.

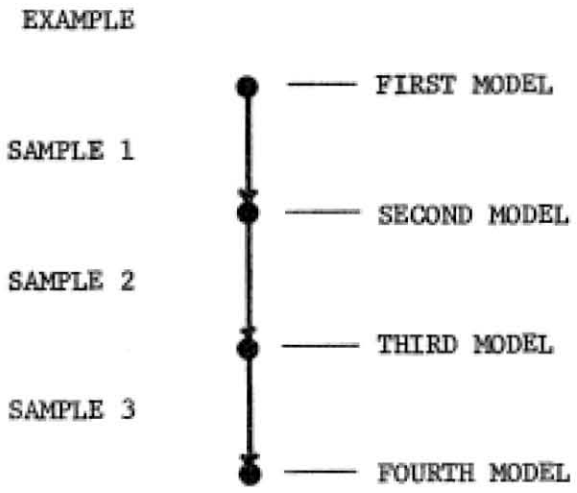


FIGURE 5-4

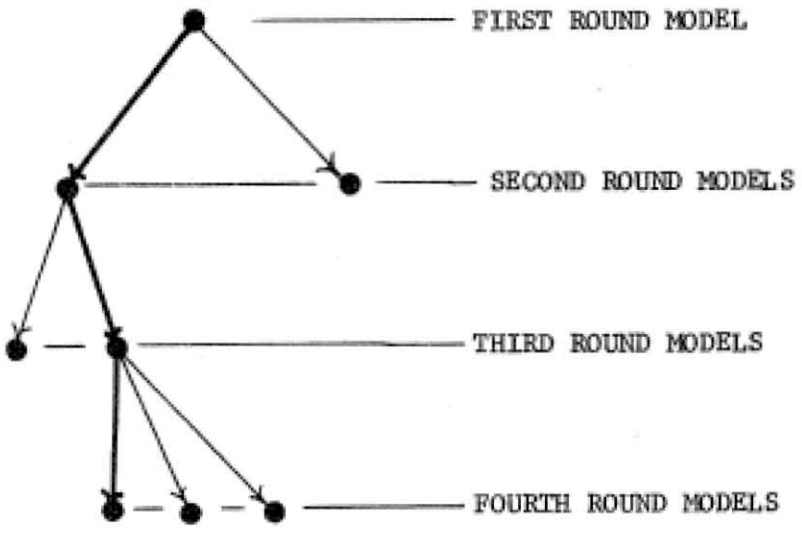


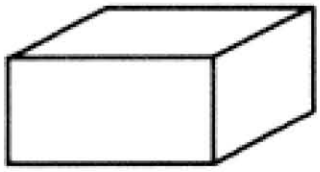
FIGURE 5-5

5.5 The Elementary Model Building Operations

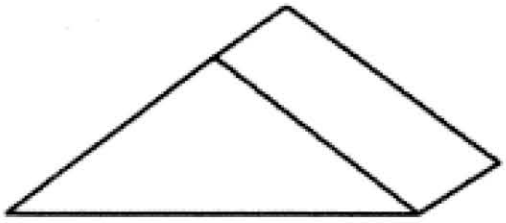
This section considers the case in which the matching program finds only one difference between the current model and a new example or near miss. The table at the end of this section summarizes the results.

5.5.1 The A-kind-of-merge: Current Model and Example

First consider the model and example in figure 5-6. Figure 5-7 shows the resulting comparison description. Only one difference is found: the object of the model points to BRICK while the object of the example points to WEDGE. But since both BRICK and WEDGE relate by A-KIND-OF to OBJECT, the a-kind-of-merge c-note occurs. Several explanations and companion responses are possible. One is that the source of the c-note may in general point to either of the things pointed to by the A-KIND-OF pointer in the two scenes. Thus the object could be either a WEDGE or a BRICK. Another possibility is that the A-KIND-OF pointers from the object do not matter at all and can be dropped from the model. Still another option and the one preferred by the program is that the object may be any member of some class in which both WEDGE and BRICK are represented. In the example one such class is simply the concept OBJECT and has already been located as the intersection of A-KIND-OF paths. The program responds by replacing the pointer in the comparison network



CURRENT MODEL



EXAMPLE

FIGURE 5-6

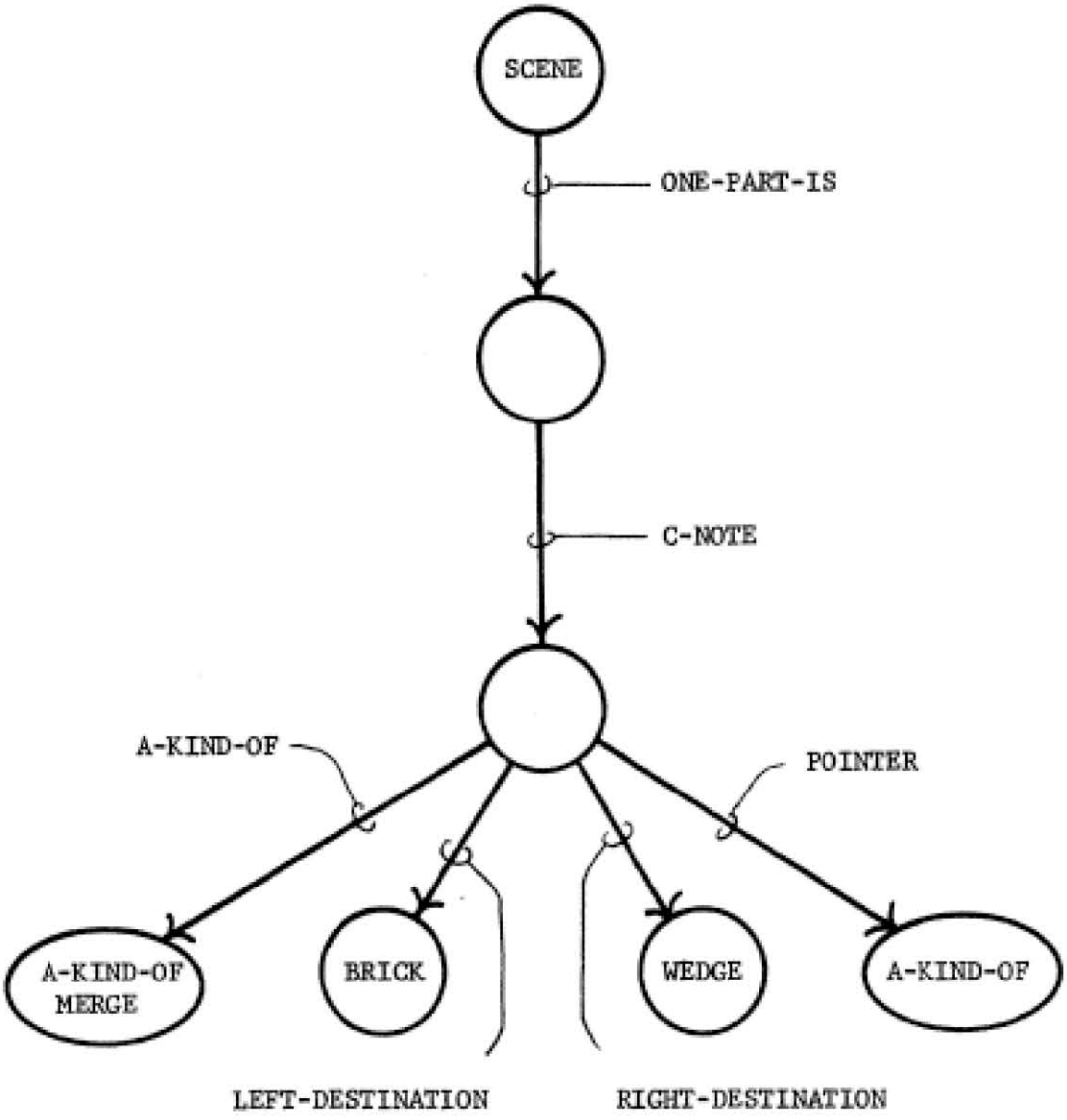

SCENE
ONE-PART-IS
C-NOTE
A-KIND-OF
POINTER
A-KIND-OF
MERGE
BRICK
WEDGE
A-KIND-OF
LEFT-DESTINATION
RIGHT-DESTINATION

FIGURE 5-7

that points to the a-kind-of-merge c-note by an A-KIND-OF pointer to the intersection or merge concept. In this case an A-KIND-OF pointer is installed between the c-note origin and the concept OBJECT. Then the altered comparison network is the new model shown in figure 5-8.

> The primary response I have selected for the machine represents a moderate stand with respect to a rather serious induction problem. I could have been more conservative and used an A-KIND-OF pointer to a concept representing the ordinary OR of BRICK and WEDGE. On the other hand, I could have been far more radical by pointing the A-KIND-OF pointer to THING, the universal class. My actual choice of pointing to the node indicated by the merge concept to be the intersection of A-KIND-OF chains seems more flexible than either of the extreems.
>
> Since the merge concept is itself defined in terms of the network's content, the generalizations made will change as the net grows. But since the appropriate response should after all depend on the universe the machine is operating in, the generalization changes are likely to be improvements. And in any case this commitment is only one among many.

5.5.2 The Supplementary-pointer C-note

Now suppose scene 1 in figure 5-9 represents the current model while scene 2 contributes as a near miss. The matching routine soon discovers that scene 1 produces a SUPPORTED-BY relation between the two objects whereas scene 2 does not. A supplementary-pointer c-note results. Of course the implication is that the concept studied requires the two objects to stand together under the support relation. Consequently, when such a supplementary-pointer c-note turns

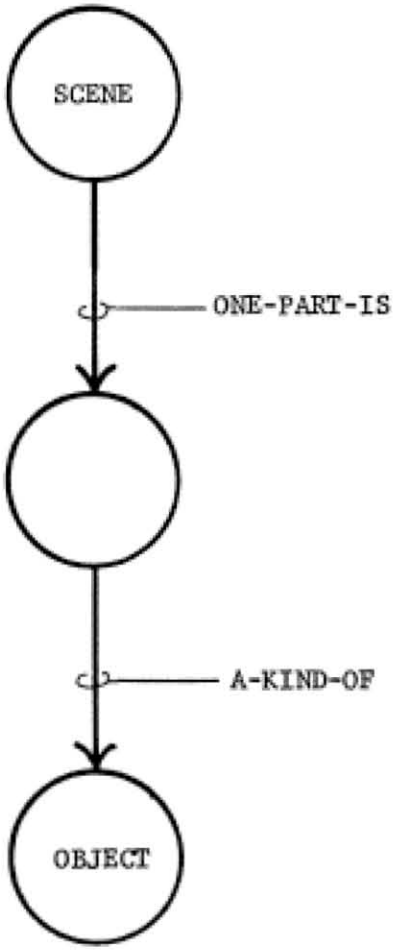
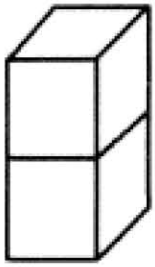


FIGURE 5-8



SCENE 1

SCENE 2

FIGURE 5-9

up, it transforms to the emphatic MUST version of the pointer involved. Thus the new model is the one in figure 5-10.

Of course the supplementary pointer can turn up in the near miss as well as in the current model. Suppose scene 1 in figure 5-9 is the near miss instead of the current model. One concludes A cannot be on B. The supplementary-pointer c-note now indicates a relation that apparently cannot hold. Appropriately, the MUST-NOT version of the supplementary pointer is substituted in and the new net appears as in figure 5-11.

5.5.3 The Must-satellite-pair C-note

Frequently comparison between the current model and a new sample displays c-notes that do not reveal any new feature, but rather result because of previous refinements in the model. Suppose, for example, that the current model has a MUST-MARRY pointer in a given location, while the sample has a MARRYS pointer. Now clearly the MARRYS pointer is appropriate in the description and the must-satellite-pair c-note consequent to matching it with MUST-MARRY should be replaced again by MUST-MARRY. Thus the emphatic form in a must-satellite-pair situation is retained and not interfered with by refinement operations attempted subsequent to its formation.

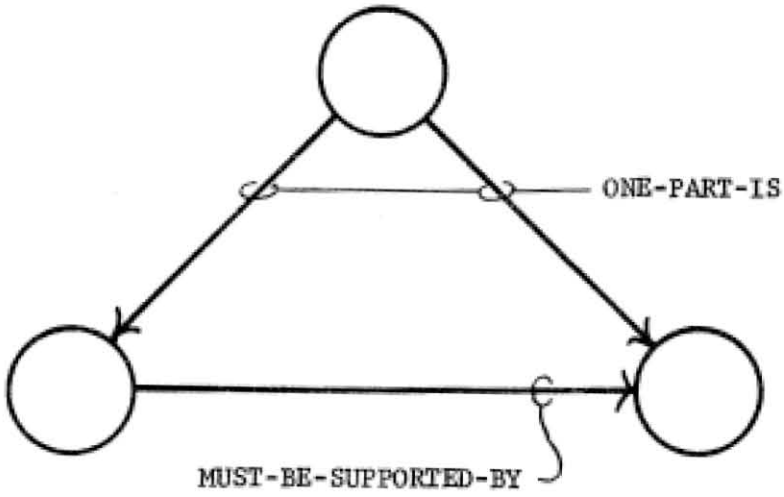


FIGURE 5-10

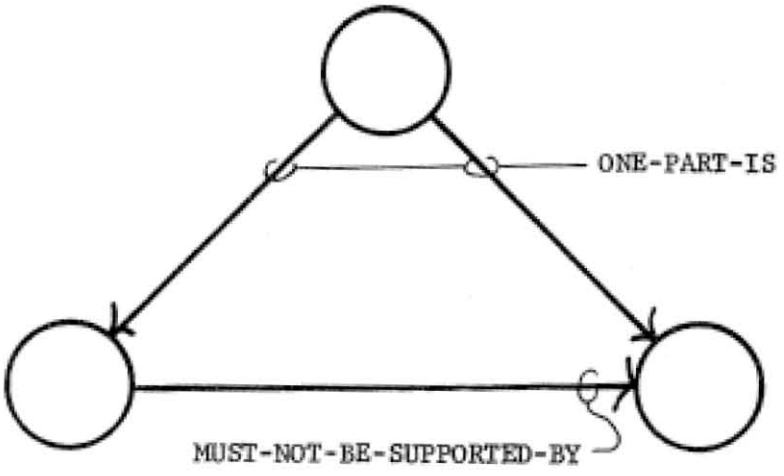
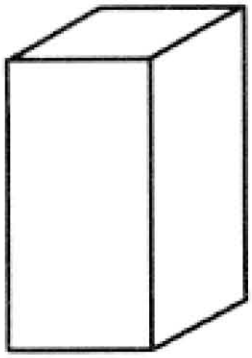


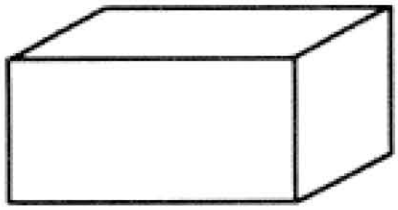
FIGURE 5-11

5.5.4 The A-kind-of-merge: Current Model and Near Miss

Sometimes a c-note offers two or more nearly equal explanations. Consider the super simple current model and near miss in figure 5-12. The c-note is an a-kind-of-merge announcing that the current model points with HAS-PROPERTY-OF to STANDING, the near miss to LYING, and both LYING and STANDING have A-KIND-OF paths to ORIENTATIONS. Now the near miss may fail either because it is lying or because it is not standing. Responding to these explanations, the model builder might replace the a-kind-of-merge c-note by a MUST-NOT-HAVE-PROPERTY-OF pointer to LYING or by a MUST-HAVE-PROPERTY-OF pointer to STANDING. Since most concepts humans discuss are defined in terms of properties rather than anti-properties, the MUST version is considered more likely.



CURRENT MODEL



NEAR-MISS

FIGURE 5-12

TABLE

ACTION OF CONCEPT GENERATOR; EXAMPLE CASE

c-note type	pointer involved	response
a-kind-of-chain	-	point to intersection with model's pointer
a-kind-of-merge	-	1. point to intersection with model's pointer 2. drop model's pointer
negative-satellite pair	-	drop model's pointer
must-be-satellite pair	-	retain model's pointer
must-not-be satellite pair	-	contradiction
supplementary-pointer or exit	negative-satellite or fundamental pointer in the model	drop model's pointer
	negative-satellite or fundamental pointer in the example	ignore
	must-be-satellite	contradiction
	must-not-be-satellite	retain model's pointer

TABLE (cont.)

ACTION OF CONCEPT GENERATOR; NEAR MISS CASE

c-note type	pointer involved	response
a-kind-of-chain	-	1. if model's node is at the end of the chain add must-not-be satellite 2. if near miss' node is at the end of the chain, use must-be satellite to model's node
a-kind-of-merge	-	1. replace model's pointer by its must-be satellite 2. replace model's pointer by must-not-be satellite of near miss' pointer
negative-satellite pair	-	replace model's pointer by its must-be satellite
must-not-be-extension pair	-	retain model's pointer
supplementary-pointer	fundamental pointer in the model	replace pointer with its must-be satellite
	fundamental pointer in the near miss	insert pointer into the using must-not-be satellite
	negative-satellite in the model	replace pointer with its must-not-be satellite
	negative-satellite in the near miss	insert pointer into model using must-be satellite

5.6 Multiple C-notes

Comparisons yielding single c-notes are rare. More often, the model builder must make sense out of a whole group of c-notes. If the comparison involves a near miss, any one of the c-notes might be the key to proper model refinement. Moreover, many of the c-notes have alternative interpretations that make further demands on executive expertise.

The model builder must therefore consider the c-notes and all the possible interpretations of each. Then it must produce the set of hypotheses that form the model tree's branches. These in turn must be ranked so that the best may be pursued first.

The case of refinement through an example is simpler than through near misses. Since none of the observed differences are sufficient to remove the example from the class, it is assumed that all of the differences found act in concert to loosen the definition embodied in the model. Consequently each c-note can be transformed independently and a new model generated by their combined action. There is no problem of deciding if one difference is more important than another.

Consequently, if all the c-notes had but one interpretation, only one new branch would be generated. The

a-kind-of-merge c-note has three possible interpretations, however, and if one such c-note occurs, it is only reasonable to create three branches instead of just one. The action on the other c-notes is the same for all three branches.

Near misses cause more severe problems. If two differences are found, either of them may be sufficient to cause the sample to be a near miss, while the other difference may be equally sufficient or merely irrelevant. If the differences have multiple interpretations or more than two differences occur, the number of possibilities explodes and the machine cannot work simply by generating an alternative for each possibility.

The model builder clearly must decide which interpretation of which differences are most likely to cause the near miss.

The machine first forms two lists: a primary list and a secondary list. Each c-note eventually ends up in one list or the other.

Now some c-notes can never make the primary list because they are of themselves insufficient to explain why a given sample is a near miss. All of these go immediately to the secondary c-note list. One example is the situation in which a pointer in the near miss corresponds in the current model to an instance of the MUST-SATELLITE version of the pointer.

A must-satellite-pair c-note results but certainly is no grounds for excluding the near miss from the class since the required pointer is in fact present. Some other explanation must be found.

The next and most obvious way to sort differences is by level. This assumes only that the differences nearer the origin of the comparison description are the more important. This certainly is a reasonable heuristic since a missing group of blocks generally impresses a human as being more important than a shape change, which in turn dwarfs a minor blemish. Consequently, the program determines the depth of the remaining c-notes which are nearest the origin of the comparison description. All those candidates found at greater depth are relegated to the secondary list.

The primary c-note list allows quick formation of little theories about why the near miss misses and what to do as a consequence. These theories are called hypotheses. A complete hypothesis specifies one c-note as the sole cause of the miss and it further specifies which interpretation of that c-note is assumed. Consequently there is a hypothesis for each interpretation of each c-note on the primary list.

The c-note specified as crucial by a hypothesis is transformed as if it were the only c-note. The other c-notes, both on the secondary and primary lists, are assumed

by the hypothesis to be insufficient cause for the near miss. Consequently as a new model is formulated according to the hypothesis, all of the c-notes but one are treated exactly as if the near miss were not a miss at all!

So far a single c-note is assumed to be the exclusive cause of the miss. Were all possible combinations considered as well, not only would the branching increase enormously, but the ranking of those branches would be difficult. Rather than face this, I have decided that only one special combination of two c-notes is ever permitted to form a hypothesis.

> In this I have exercised what one might call the first heuristic of science: Begin with the linear model, the one that assumes all things act independently; then consider interactions as necessary. I next discuss a particular case in which it does seem necessary to consider the joint action of two differences. It would be unreasonable, however, to try for a general method for handling multiple differences. In science as a whole, each particular method for treating interacting effects is usally a major problem in itself and over-ambitious search for completely general methods is of low utility when premature.
>
> Further justification for my appproch lies in certain observations of Piaget's that indicate that children seem to pay sharp attention to only a single feature at any one time [3]. In comparing volumes, for example, they use mainly height. Yet, in spite of using what appear to be linear comparisons, these same children can learn physical concepts with a talent far in excess of my goal for this thesis.

Hypotheses based on two contributing c-notes are added to the hypothesis list only when two c-notes with nearly

identical descriptions occur. Consider figure 5-13. Since exactly the same thing characterizes both blocks in the near miss, there is no particular reason to suppose that one difference should be singled out. Consequently a third hypothesis is formed, namely that both differences act cooperatively. This additional hypothesis takes precedence over the two hypotheses that consider the differences separately. It seems heuristically sound that coincidences are significant. The machine creates new models with such hypotheses by transforming both of the specified c-notes in the miss-explanation mode.

5.7 Contradictions and Backing Up

By now one may wonder why the program should deal with alternatives to the main line of model development at all. To be sure, maximum likelihood assumptions may be wrong, but then how could the machine ever know when such a decision is an error? The answer is that the main line assumptions may lead to contradiction crises which in turn cause the model building program to retreat up the tree and attempt model development along other branches.

Consider again the very simple situation presented in figure 5-14. The current model and the near miss combination generate an a-kind-of-merge c-note for which the priority interpretation is that examples of the concept must

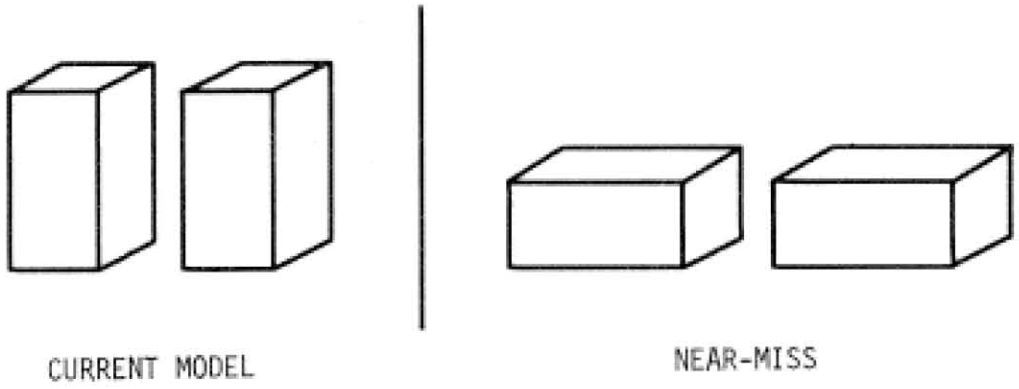
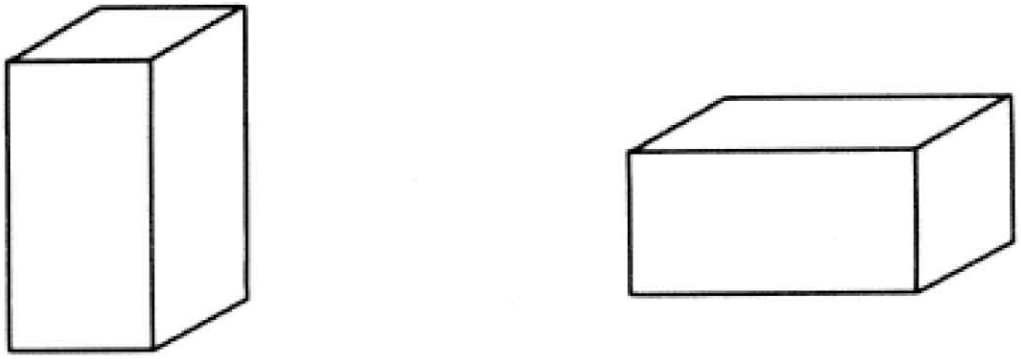


FIGURE 5-13



CURRENT MODEL

NEAR-MISS

FIGURE 5-14

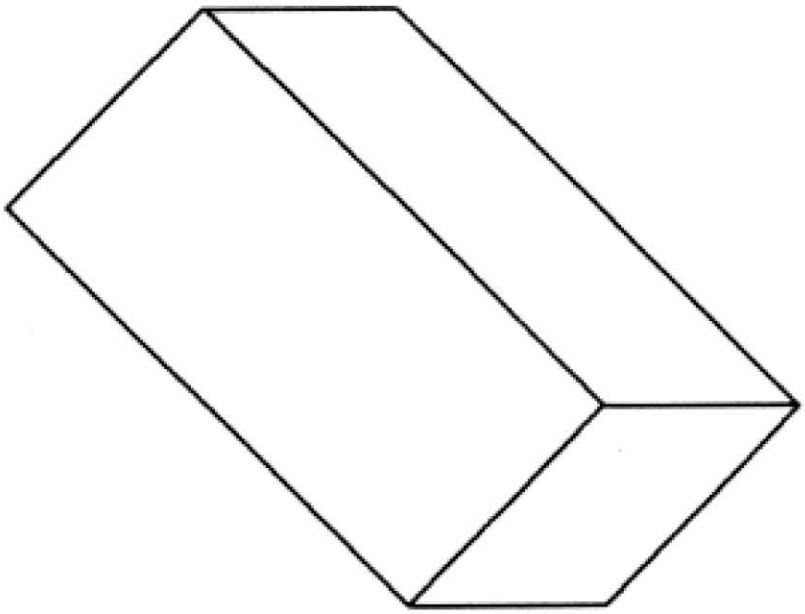


FIGURE 5-15

be standing. The alternative, that examples must not be lying, causes a side branch in the model development tree. But suppose one really wants the concept to exclude lying but not insist on standing. Showing the machine the example in figure 5-15 does the job. The tilted brick certainly is not standing and its description has no HAS-PROPERTY-OF pointer to STANDING. Yet the current model has a MUST-HAVE-PROPERTY-OF pointer to STANDING. This is a contradictory situation.

When contradictory situations occur, the program assumes it has made an incorrect choice somewhere, closes the branch to further exploration, and backs up one level to select another alternative if any are available there. If no alternatives are available, the program backs up still more levels until either an unexplored alternative is found, or the top level is reached. If the top level is reached with no other options found, the program succumbs and admits failure. More often an acceptable unexplored alternative is soon found and an effort is made to extend the model tree down that branch. Of course, the first alternative a contradiction causes to be explored may itself lead to contradiction. Back up then starts from the new contradiction and proceeds as before.

In the case at hand, an alternative is found and the must-not-be-lying interpretation of the comparison between

the scenes in figure 5-14 leads to a new intermediate model. This in turn is refined by the scene of figure 5-15 which originally caused the contradiction on the former main line. No contradiction occurs on the new path because the MUST-NOT-HAVE-PROPERTY-OF - LYING combination of the intermediate model has nothing to clash with in the example. Indeed the new example lends no new information to model development along this path, the model being the same before and after comparison. The new example served solely to terminate development of an improper path in the model development tree.

5.8 Other Backing Up Possibilities

Many possible refinements to the elementary backing up procedure invite attention. For one thing there are other reasons why the learning program might want to back up. In addition to the situation of direct contradiction, attention should move back up the model tree if there are so many differences between the current model and the sample that hopeless confusion is suggested. The cause of such confusion is likely to lie in the selection of a wrong branch at some higher point in the model tree. Similarly retreat is in order if the program is forced to propose an unlikely explanation to account for observed differences.

Right now the learning program backs up level by level,

blindly exploring all possible paths from one branch point before backing up to the next higher branch point. It would be better if attention could move directly to the point in the tree where the problem began. There a better alternative could be elected and learning could more likely procede in an orderly way.

Certainly the selection of the appropriate point is easy in the case of direct contradiction. As explained before, these situations occur when some relation is found to be essential at some point in model development only to be absent from some subsequent example of the concept. The crucial point is the place where the decision was made that the relation was essential. This is the point where attention should go and an alternative explanation should be sought.

6 Some Generated Concepts

6.1 Physical Models and Functional Models

In this chapter I explore some of the properties of the model generator through a series of examples. In the course of this discussion, words like house, arch, and tent occur frequently as they are convenient names for the ideas the machine assimilates. Be cautioned, however, to avoid thinking of these entities in terms of functional definitions. To a human, an arch may be something to walk through, as well as an appropriate alignment of bricks. And certainly, a flat rock serves as a table to a hungry person, although far removed from the image the word table usually calls to mind.

But the machine does not yet know anything of walking, residing, or eating, so the programs discussed here handle only some of the physical aspects of these human notions.

> There is nothing mystical about this. There is no inherent obstacle forbidding the machine to enjoy functional understanding. It is a matter of generalizing the machine's descriptive ability to acts and properties required by those acts. Then chains of pointers can link TABLE to FOOD as well as to the physical image of a table, and then the machine will be perfectly happy to draw up its chair to a flat rock with the human, given that there is something on that table which it wishes to eat.

5.2 The House

Figure 6-1 illustrates what house means here. Basically the scene is just one wedge on top of one brick. But lacking human experience, this one picture is insufficient to convey much of the notion house to the machine. The model builder must be used, and it must be permitted to observe other samples.

Suppose the model builder starts with the scene in figure 6-1. Then its description generation apparatus contributes the network which serves as the first unrefined, unembellished model. (figure 6-5) Now suppose the scene in figure 6-2, a near miss, is the next sample. Its net is that shown in figure 6-6. The only difference is the supplementary pointer SUPPORTED-BY. Glancing at the table of section 5.5, it is clear that the overall result is conversion of the SUPPORTED-BY pointer in the old model to MUST-BE-SUPPORTED-BY in the new model. Thus the new model is that of figure 6-7. Figure 6-8 shows the current model development tree.

Much is yet to be learned. For one thing, the top object certainly must be a wedge. Showing the machine the near miss of figure 6-3 conveys this point immediately. Similarly the near miss of figure 6-4 makes the brick property of the bottom object mandatory. But notice that

HOUSE

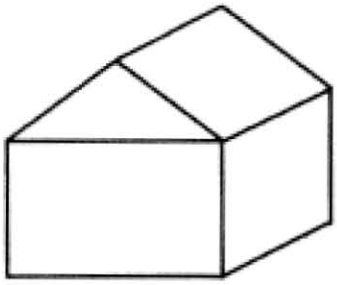


FIGURE 6-1

NEAR MISS

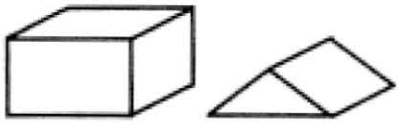


FIGURE 6-2

NEAR MISS

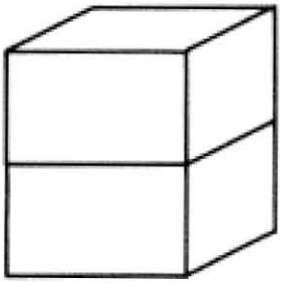


FIGURE 6-3

NEAR MISS

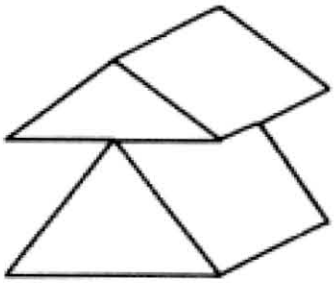


FIGURE 6-4

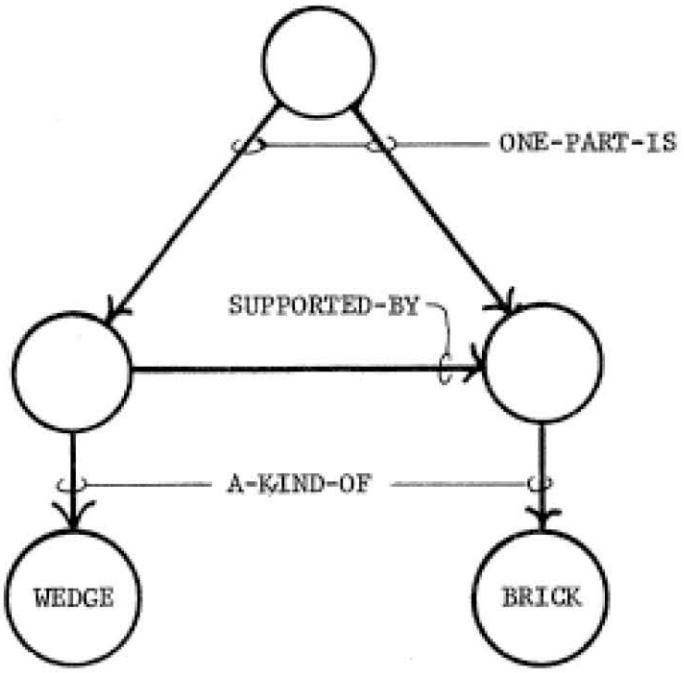


FIGURE 6-5

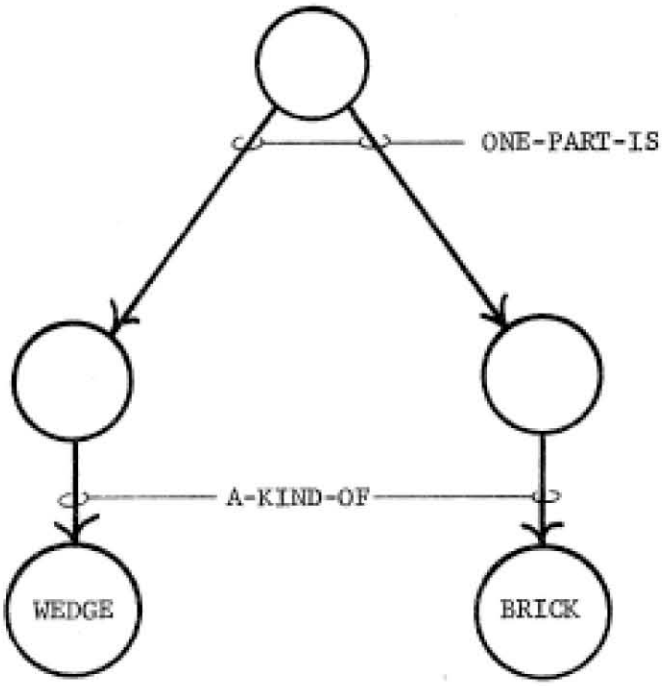


FIGURE 6-6

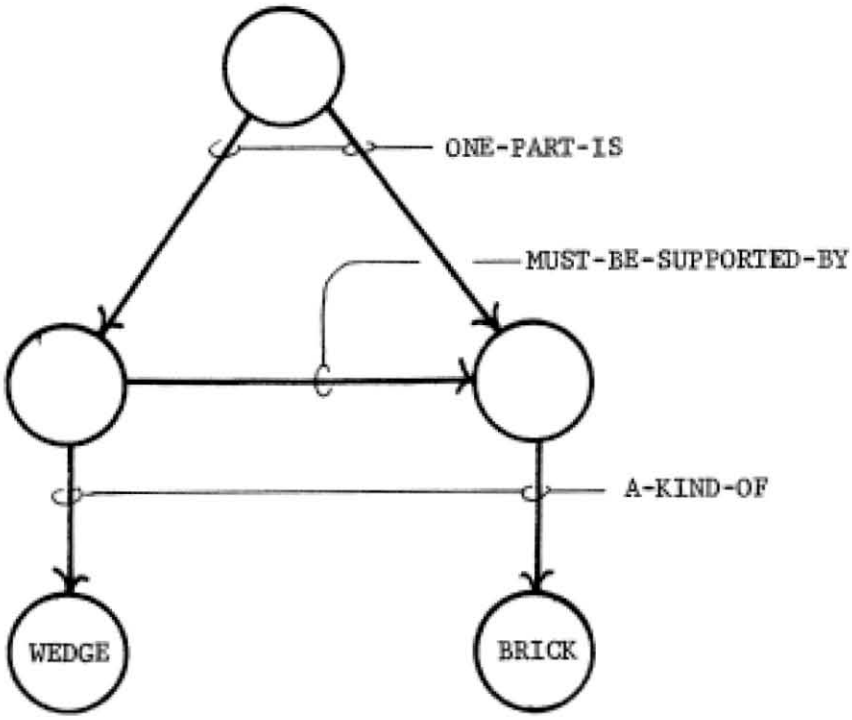


FIGURE 6-7

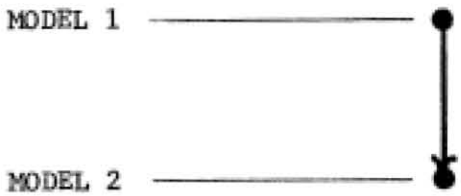


FIGURE 6-8

both of these steps cause bifurcation of the model tree. The reason is that the machine cannot be completely sure the miss occurs because the old property is lost or because the new property is added. The program prefers the old-property-is-lost theory and moves down the corresponding branch unless contradicted. In both of these situations, the preferred theory is correct resulting in the final model and tree shown in figure 6-9 and figure 6-10.

6.3 The Pedestal

Development of a pedestal model proceeds much as does the house with only two essential differences. First, the top object must be a brick rather than a wedge. Second, the upper object must not marry the lower. The scene in figure 6-11 yields the starting model. Figure 6-12 forces the top object to be a brick while figure 6-13 forces the bottom object to be a brick as well. Figure 6-14 emphasizes support. And finally, figure 6-15 forbids the MARRYS relation.

6.4 The Tent

Think of the tent as two wedges, marrying each other. As such it illustrates the handling of two similar differences simultaneously.

Suppose the base model is the description of the scene in figure 6-16 and the first sample is the near miss in

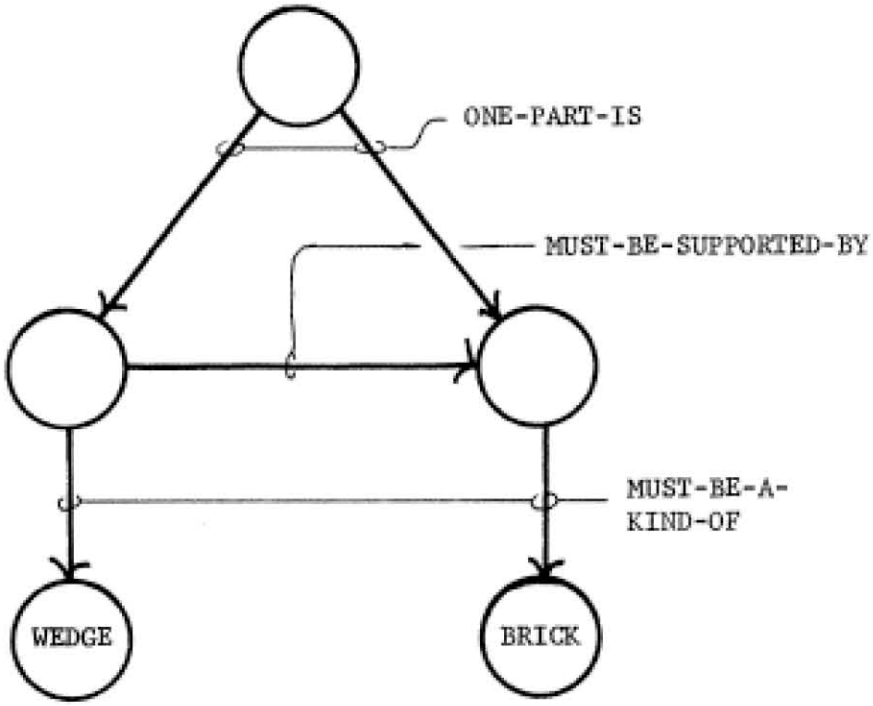


FIGURE 6-9

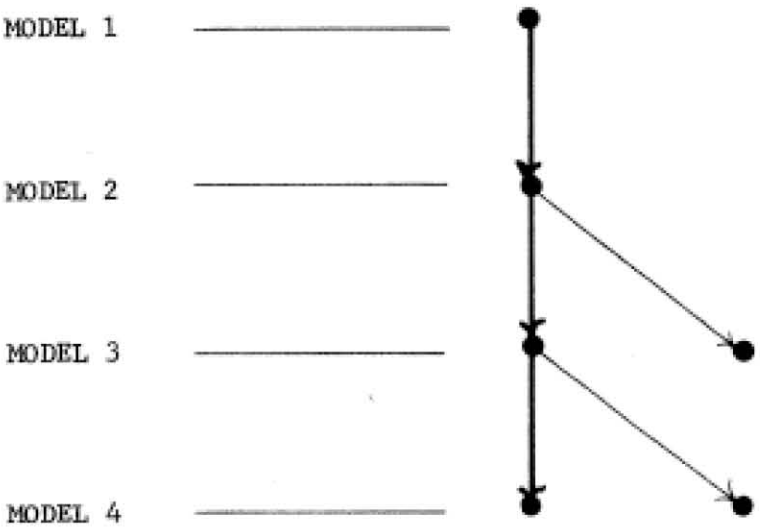


FIGURE 6-10

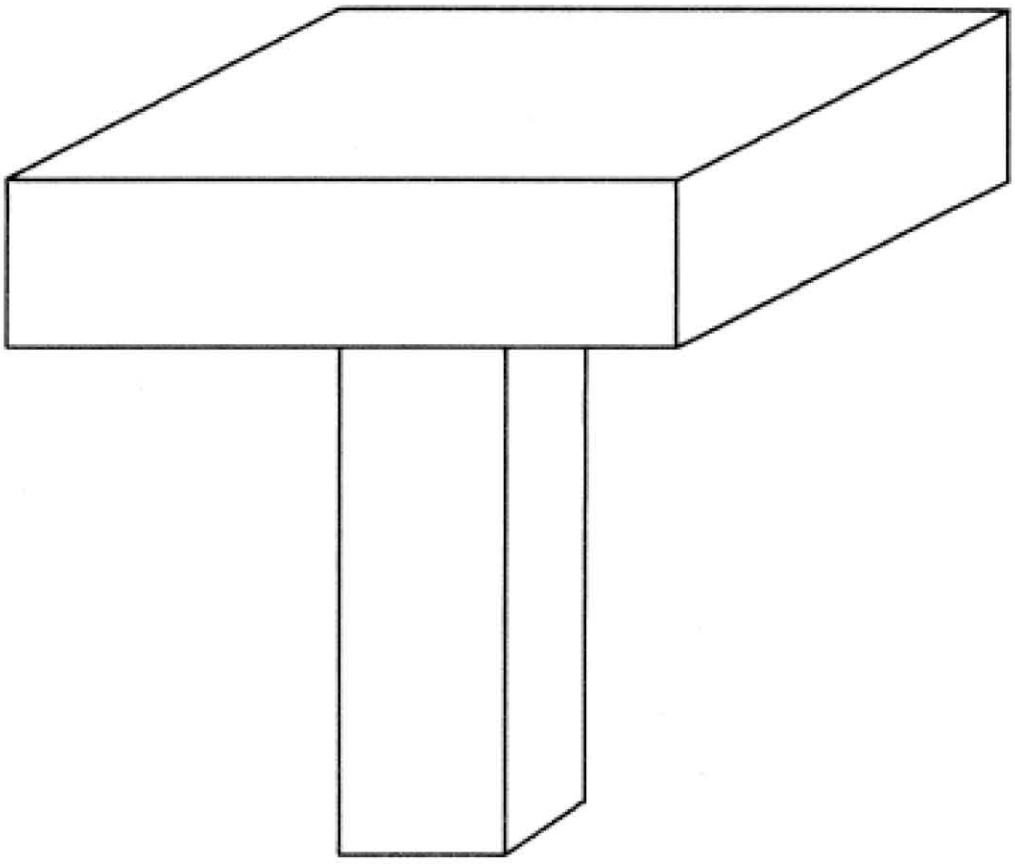


FIGURE 6-11: PEDESTAL

NEAR MISS

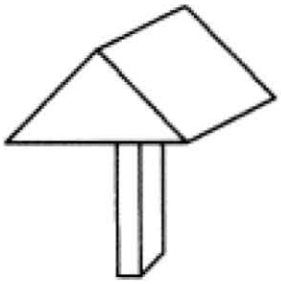


FIGURE 6-12

NEAR MISS

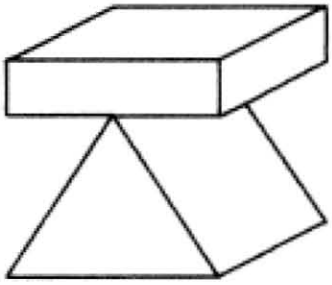


FIGURE 6-13

NEAR MISS

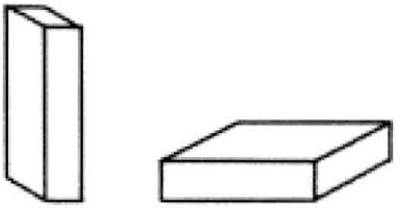


FIGURE 6-14

NEAR MISS

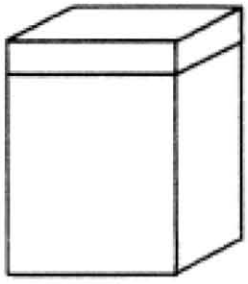


FIGURE 6-15

TENT

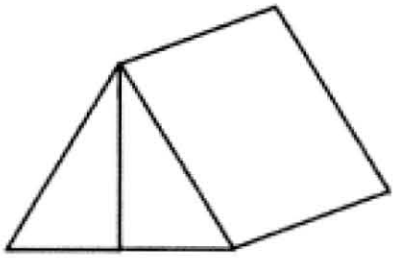


FIGURE 6-16

NEAR MISS

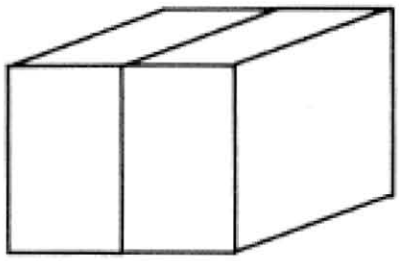


FIGURE 6-17

NEAR MISS

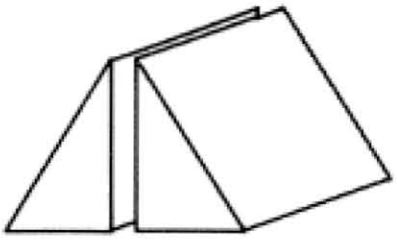


FIGURE 6-18

figure 6-17. Two a-kind-of-merge c-notes result, one from each of the two objects because they are bricks not wedges. Since they differ only in source, the hypothesis that both act together is the priority one, which leads to the result in figure 6-19. Now this is complemented by the near miss in figure 6-18 which informs the machine of the importance of the MARRYS relation. Again dual c-notes announce the loss of a pair of MARRYS pointers and twin MUST-MARRY pointers are installed. (figure 6-20)

6.5 The Arch

The arch involves a mixture of the elements seen in the previous examples. Because of the wider variety of differences encountered, it produces a bushy tree and a challenge to routines that select priority hypotheses.

The scene in figure 6-21 forms the first model. Combining this with the scene in figure 6-22, the machine deduces that the MARRYS relations between the top and the supports are not crucial.

Next the near miss of figure 6-23 indicates that the support relations are crucial. Again, both new MUST-BE-SUPPORTED-BY pointers are handled jointly, and are installed at once.

The machine learns perhaps the most important fact from the near miss in figure 6-24. Here the two supports touch,

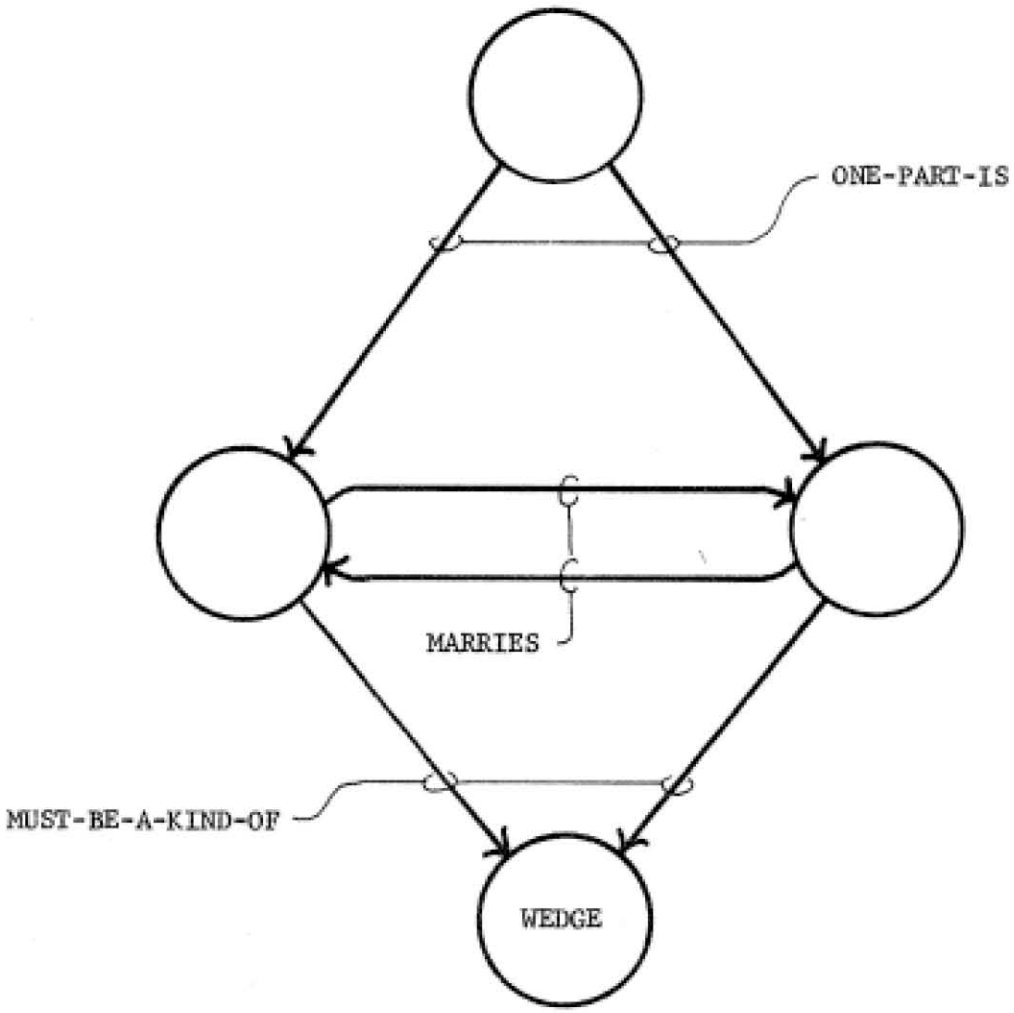


FIGURE 6-19

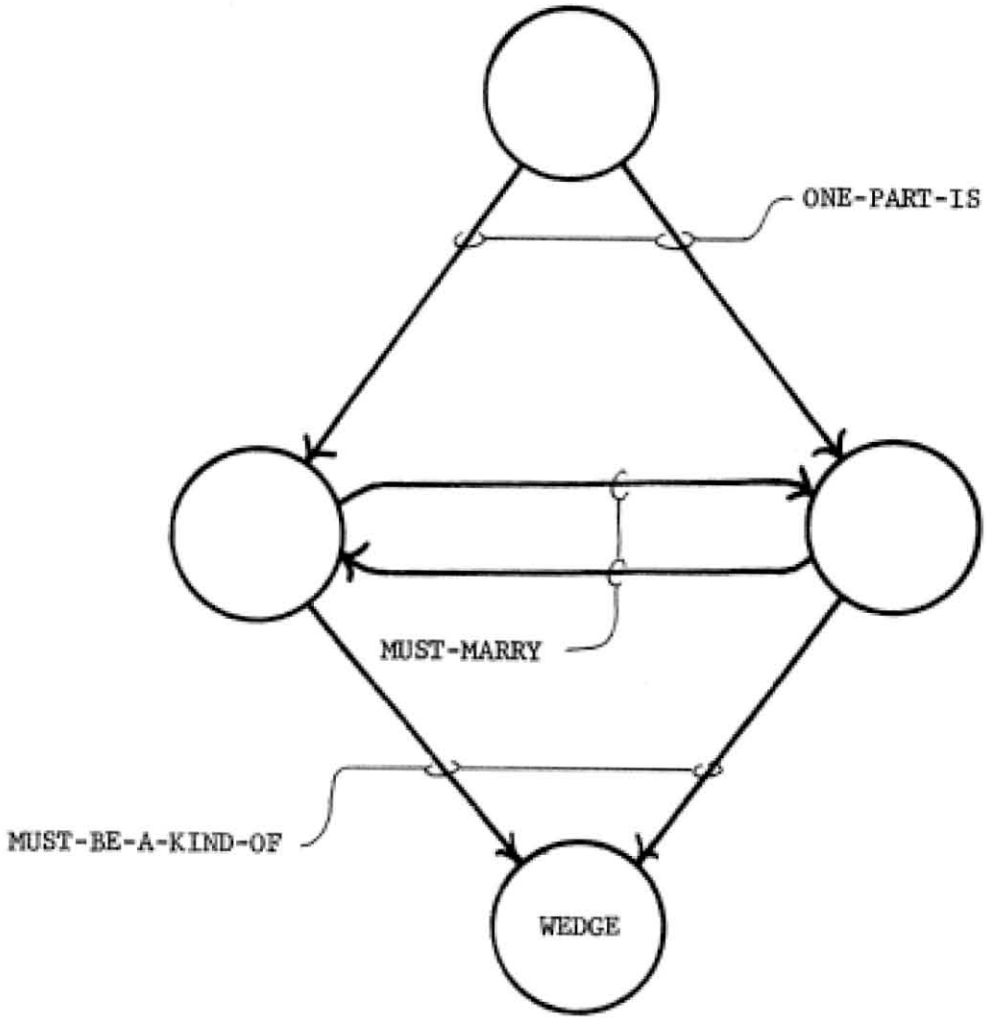


FIGURE 6-20

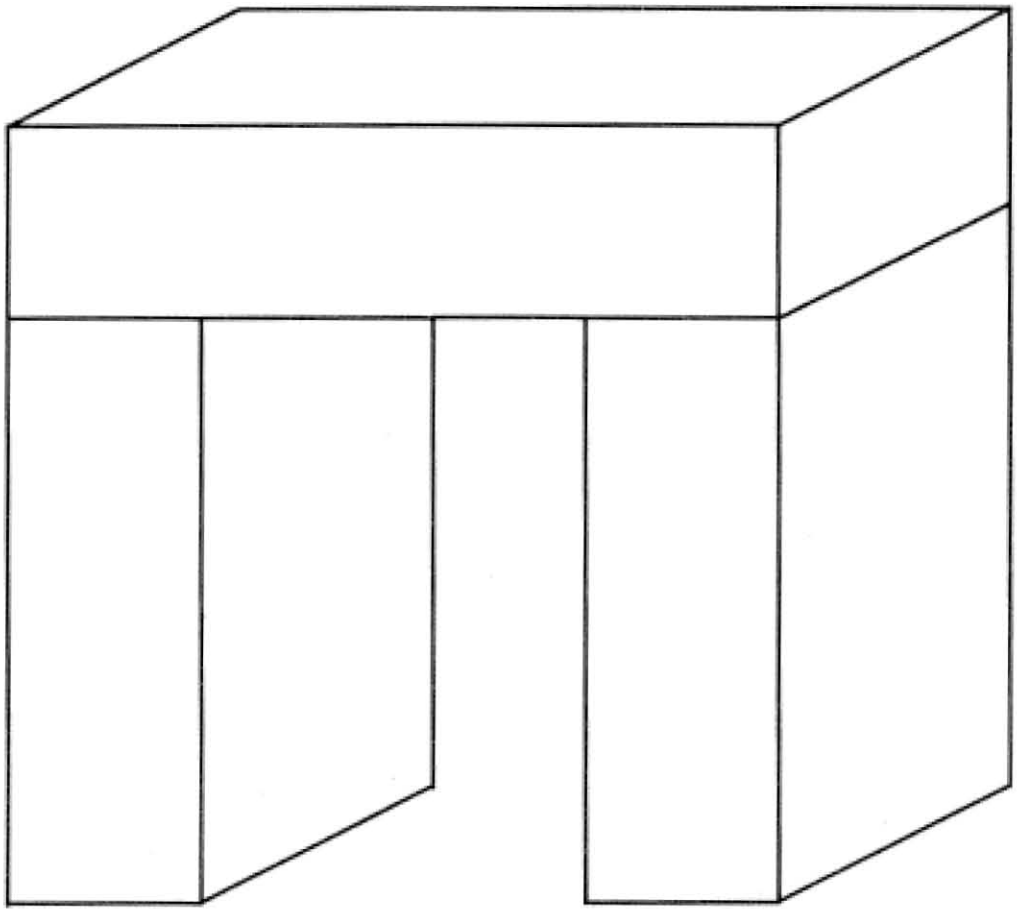


FIGURE 6-21: ARCH

ARCH

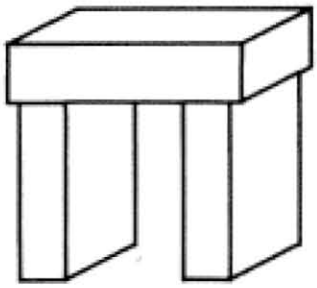


FIGURE 6-22

NEAR MISS

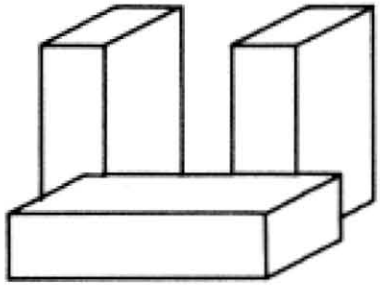


FIGURE 6-23

NEAR MISS

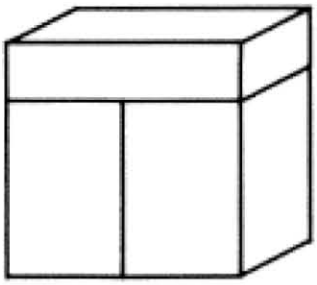


FIGURE 6-24

ARCH

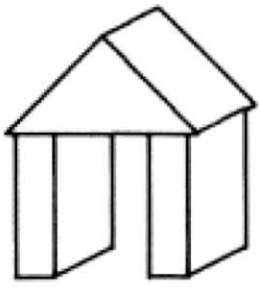


FIGURE 6-25

supplying two MARRYS pointers to the description. This cannot be allowed. Responding, the machine inserts MUST-NOT-MARRY pointers between the two supports in the model.

> Some may think that in asserting the MUST-NOT-MARRY relations, the machine overlooks what they consider the real principle, that of a hole or passage. But for a child building with blocks, to have a hole and to have two non-touching supports are very nearly the same idea. Consequently the machine's opinion seems adequate for the moment. Experiments such as these may help to expose exactly what kinds of network relations are adequate for a model of human thinking, from infant to adult.

Finally, the top object is not necessarily a brick. The sample in figure 6-25 teaches the machine that anything in the class OBJECT will do, since OBJECT lies but one step removed by an A-KIND-OF pointer from both WEDGE and BRICK.

6.6 The Wedge

The capabilities of the model builder certainly extend beyond the level of object configurations, whose descriptions allow the machine to learn about scenes. Here the development of the wedge model illustrates the point.

Given the wedge in figure 6-26, the description generated is that of figure 6-27.

Next, comparison with a brick establishes a MUST-BE-A-KIND-OF pointer to TRIANGLE. (figure 6-28)

But now suppose the partly occluded object in figure 6-29 is compared first with the current model of wedge

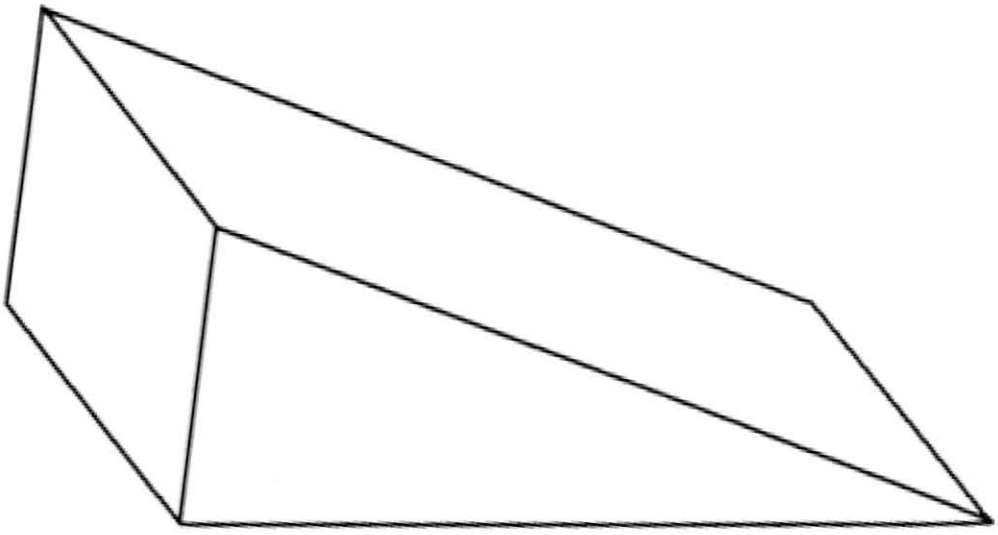


FIGURE 6-26: WEDGE

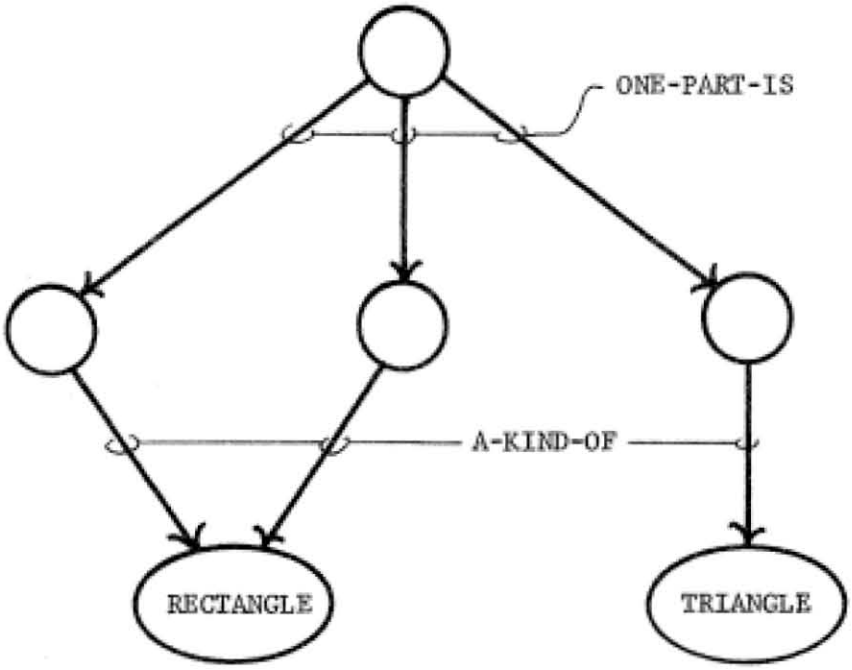


FIGURE 6-27

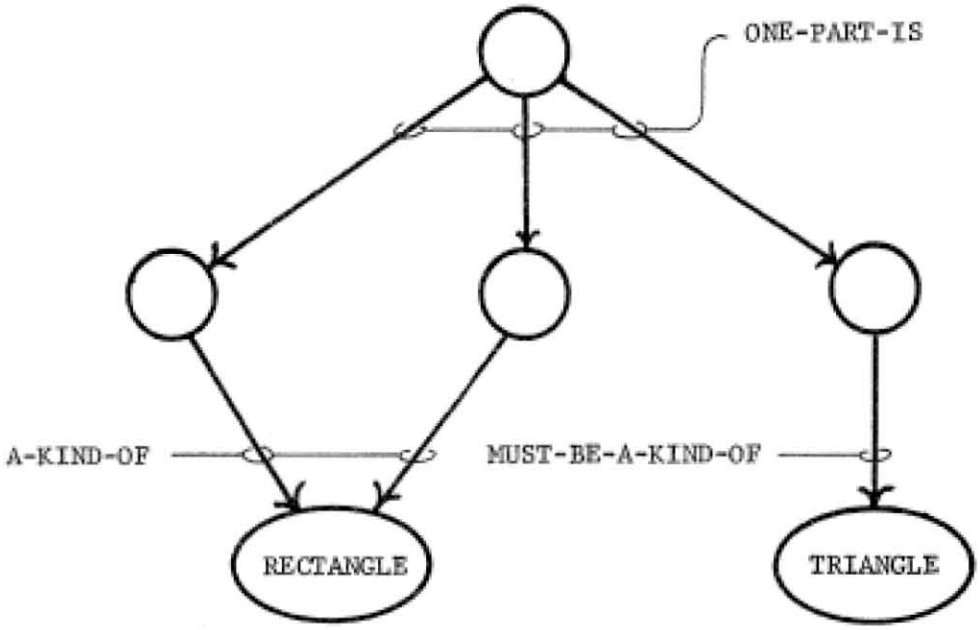


FIGURE 6-28

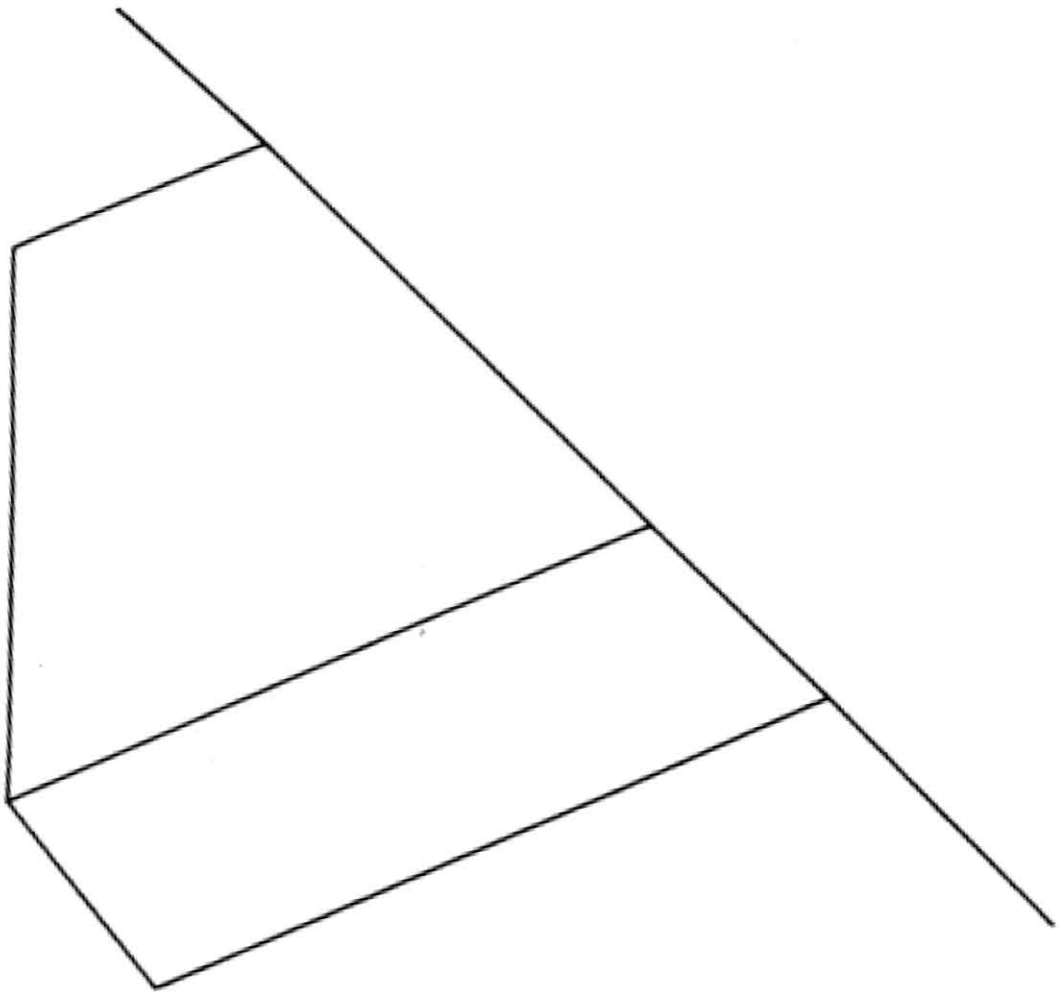


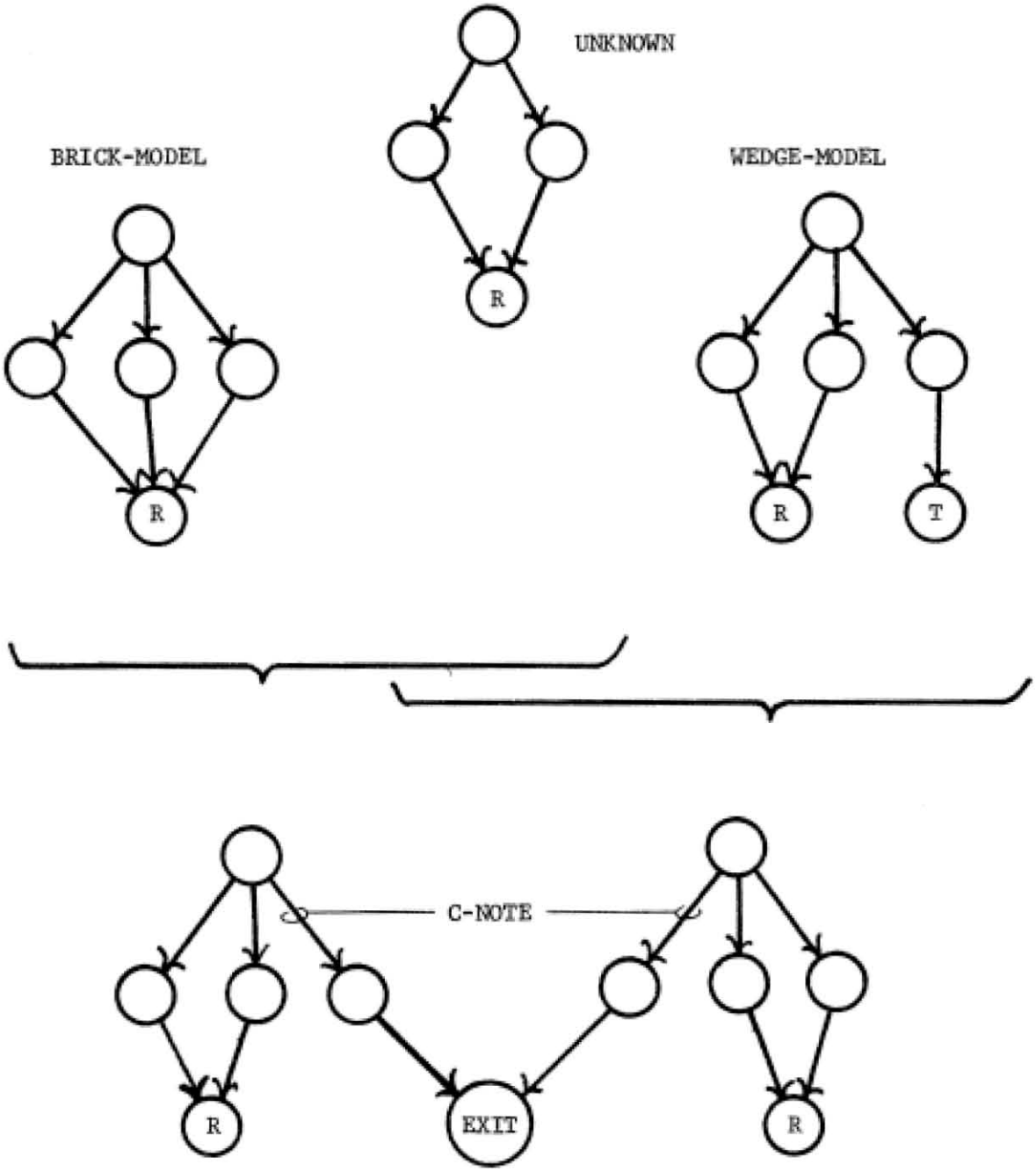
FIGURE 6-29: NOT A WEDGE

and then with a description of a brick. As figure 6-30 illustrates, the surprising result is that both comparisons have nearly the same descriptions. In both cases two rectangles are matched and a third side left unmatched. In one case the unmatched side is another rectangle, and in the other, it is a triangle. But there is as yet no way to prefer one match above the other!

The problem is a little involved. No severe mismatch is evident to the difference description evaluation program because the MUST-BE-A-KIND-OF pointer is anchored to a node that is not matched. The model as it stands asserts firmly that if three sides are seen, one of them must be a triangle, but it does not assert that such a third side must be present.

This may seem to be a bug at first, but the problem is really the machine's lack of experience. So far only two configurations have contributed to the model. Consider the result of refining the model with the scene in figure 6-29 which is causing the trouble. It is a near miss. But the difference between it and the current model is an exit c-note to the triangular side. The model builder perceives the need for an emphatic pointer and ONE-PART-MUST-BE is inserted. (figure 6-31)

Now if this model is compared again with the scene that



R = RECTANGLE

T = TRIANGLE

FIGURE 6-30

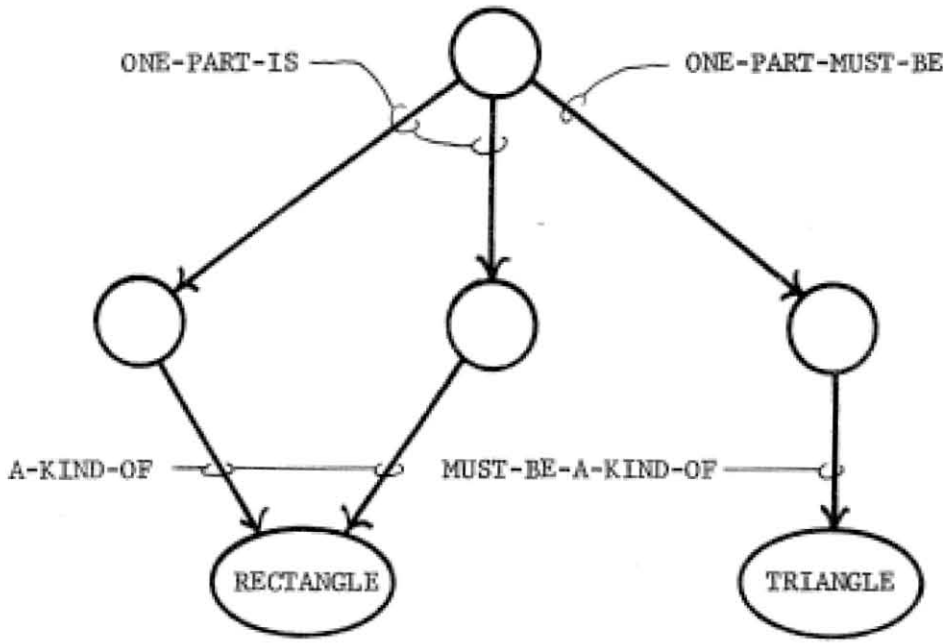


FIGURE 6-31

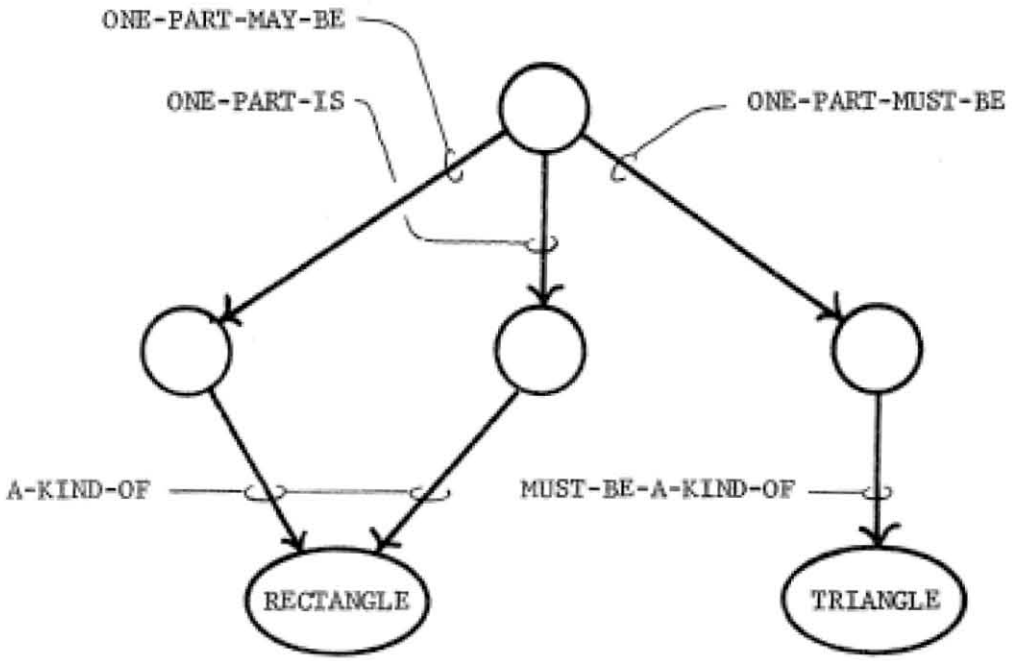


FIGURE 6-33

just refined the model, the one in figure 6-29, the result is an exit c-note bearing an emphatic pointer, ONE-PART-MUST-BE. Such a c-note strongly suggests bad match to the evaluation program, and the apparent inadequacy disappears.

The scene in figure 6-32 establishes a final refinement. This wedge shows only two sides. After a bit of thinking, the program decides one of the two rectangular sides is optional and produces its final model. (figure 6-33)

6.7 The Composite Column

When a concept involves groups of objects, the model generation problem really is no more difficult. It just becomes a matter of concentrating on relationships between the typical members of the groups studied.

Consider the notion of the composite column, hereafter referred to simply as the column. Figure 6-34 shows such a column and figure 6-40 shows part of the corresponding descriptive network. This description is gradually transformed into a reasonable model in the following way:

Figure 6-35, with its bricks askew but otherwise the same, introduces the MUST-MARRY pointer. Figure 6-36, made of wedges instead of bricks, relaxes the inclination toward bricks. And figure 6-37 causes replacement of SUPPORTED-BY by MUST-BE-SUPPORTED-BY.

Next, figure 6-38 contributes the most important part of

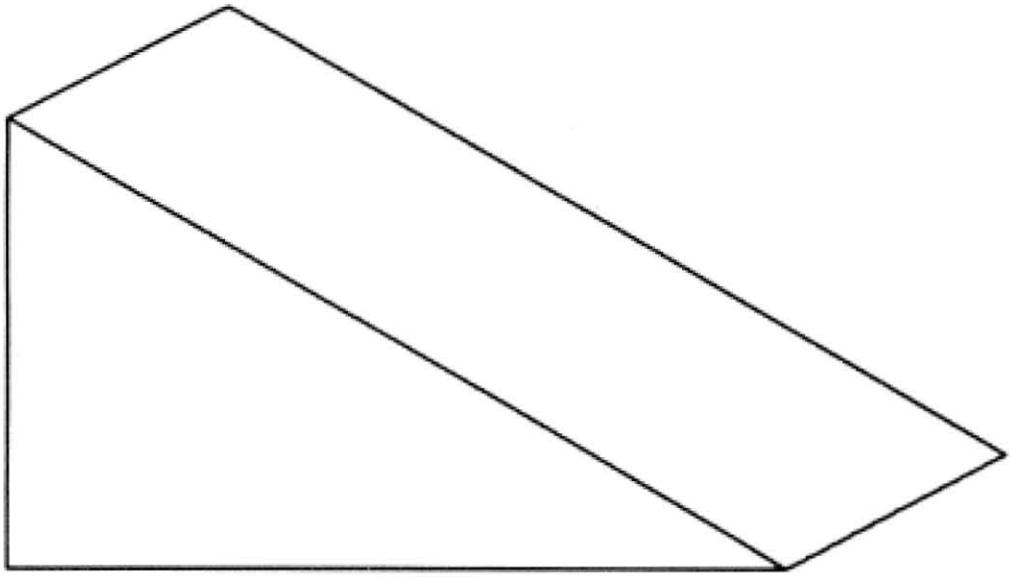


FIGURE 6-32: WEDGE

COLUMN

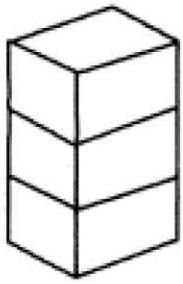


FIGURE 6-34

NEAR MISS

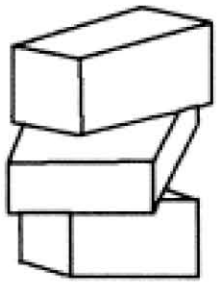


FIGURE 6-35

COLUMN

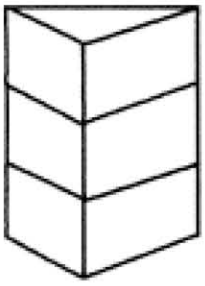


FIGURE 6-36

NEAR MISS

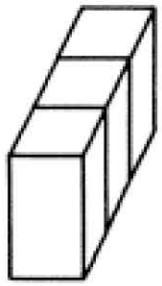


FIGURE 6-37

NEAR MISS

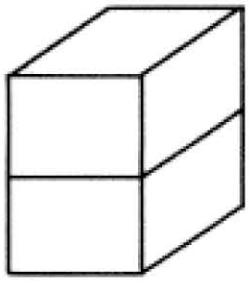


FIGURE 6-38

COLUMN

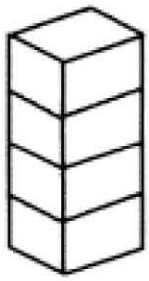


FIGURE 6-39

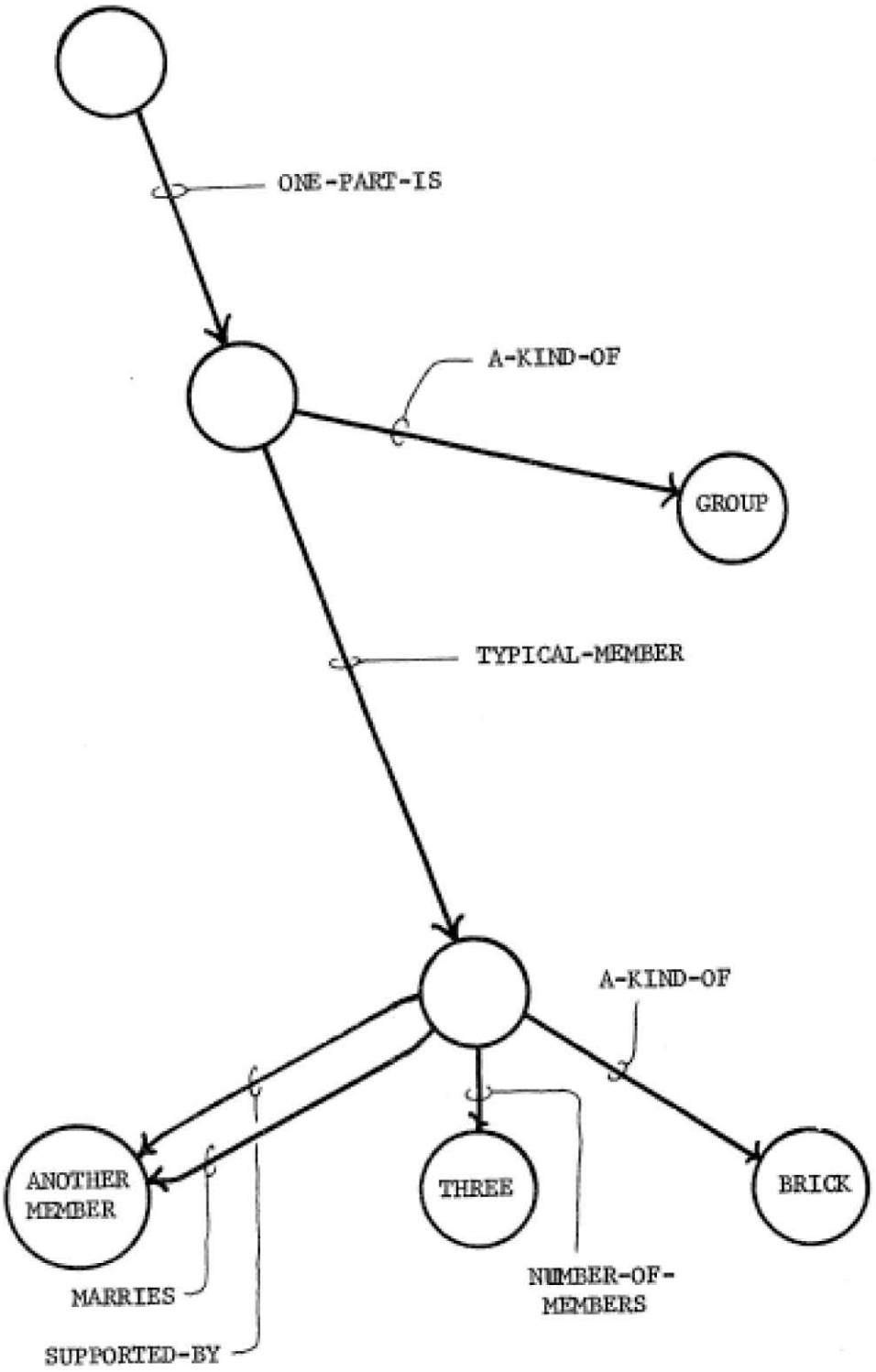

ONE-PART-IS
A-KIND-OF
GROUP
TYPICAL-MEMBER
A-KIND-OF
ANOTHER MEMBER
THREE
BRICK
MARRIES
SUPPORTED-BY
NUMBER-OF-MEMBERS

FIGURE 6-40

the model, the part that demands a group. The scene in figure 6-38 consists of only two objects and is not grouped because the grouping program requires a minimum of three objects. The resulting c-notes reflect the mandatory need for a group by way of a ONE-PART-MUST-BE pointer to the node representing the group.

Finally, figure 6-39 generalizes the model in an important way because its typical-member node differs from the model's principally because a different number of objects is indicated. In one case the NUMBER-OF-MEMBERS pointer points to 3; in the other, 4. But since both 3 and 4 are integers and have A-KIND-OF pointers to INTEGER, when the comparison between the two is made an a-kind-of-merge c-note results. The next model consequently has a NUMBER-OF-MEMBERS pointer to INTEGER, rather than 3. Now columns may have any number of objects greater than two.

Figure 6-41 displays the overall result.

5.8 The Arcade

The problem of learning about the arcade shown in figure 6-42 adds another interesting dimension to the model generation problem. Nothing new is needed in developing a sequence of models for the arcade, with one important exception:

The arcade is a conglomeration of substructures, rather

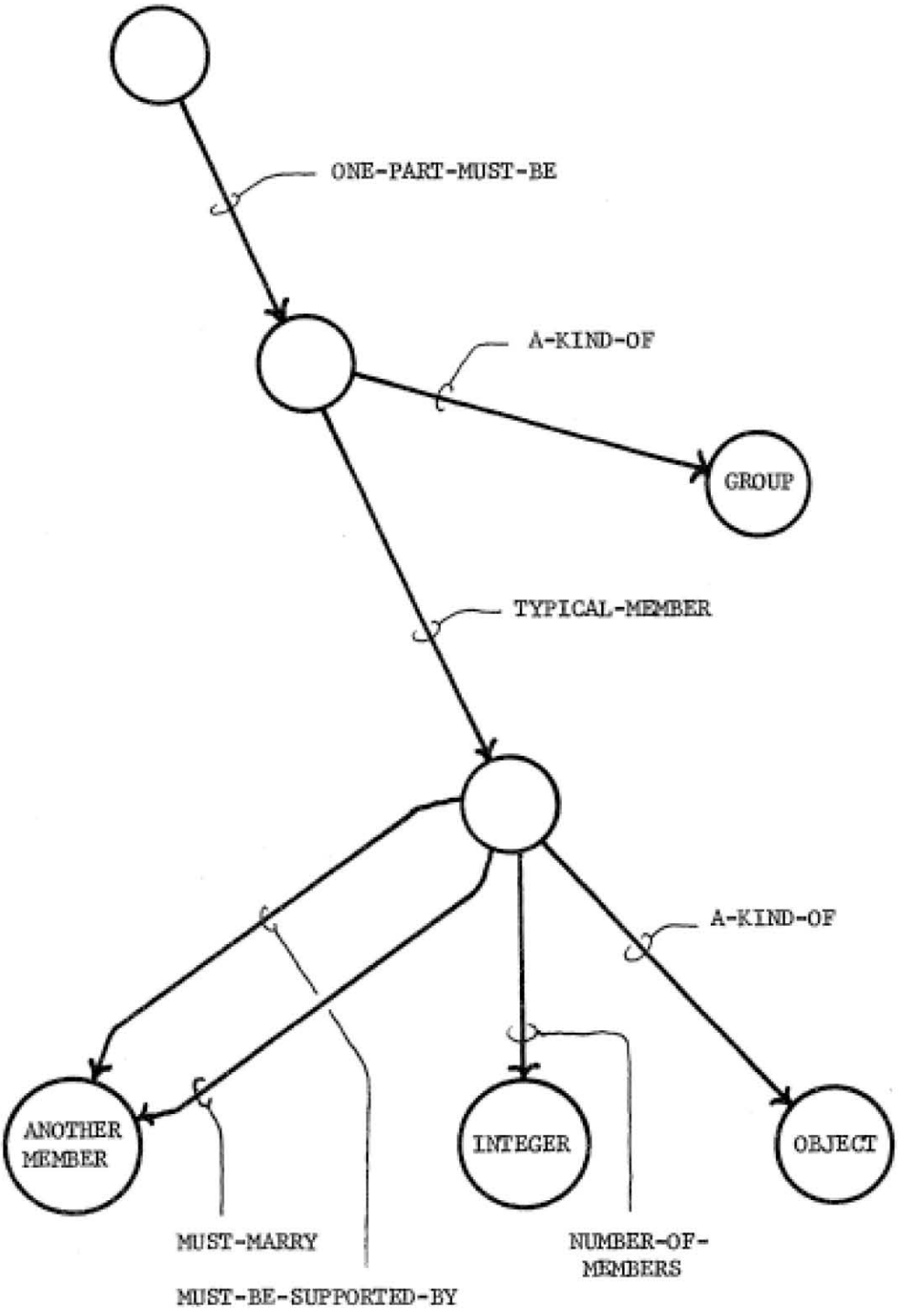

ONE-PART-MUST-BE
A-KIND-OF
GROUP
TYPICAL-MEMBER
A-KIND-OF
ANOTHER MEMBER
INTEGER
OBJECT
MUST-MARRY
MUST-BE-SUPPORTED-BY
NUMBER-OF-MEMBERS

FIGURE 6-41

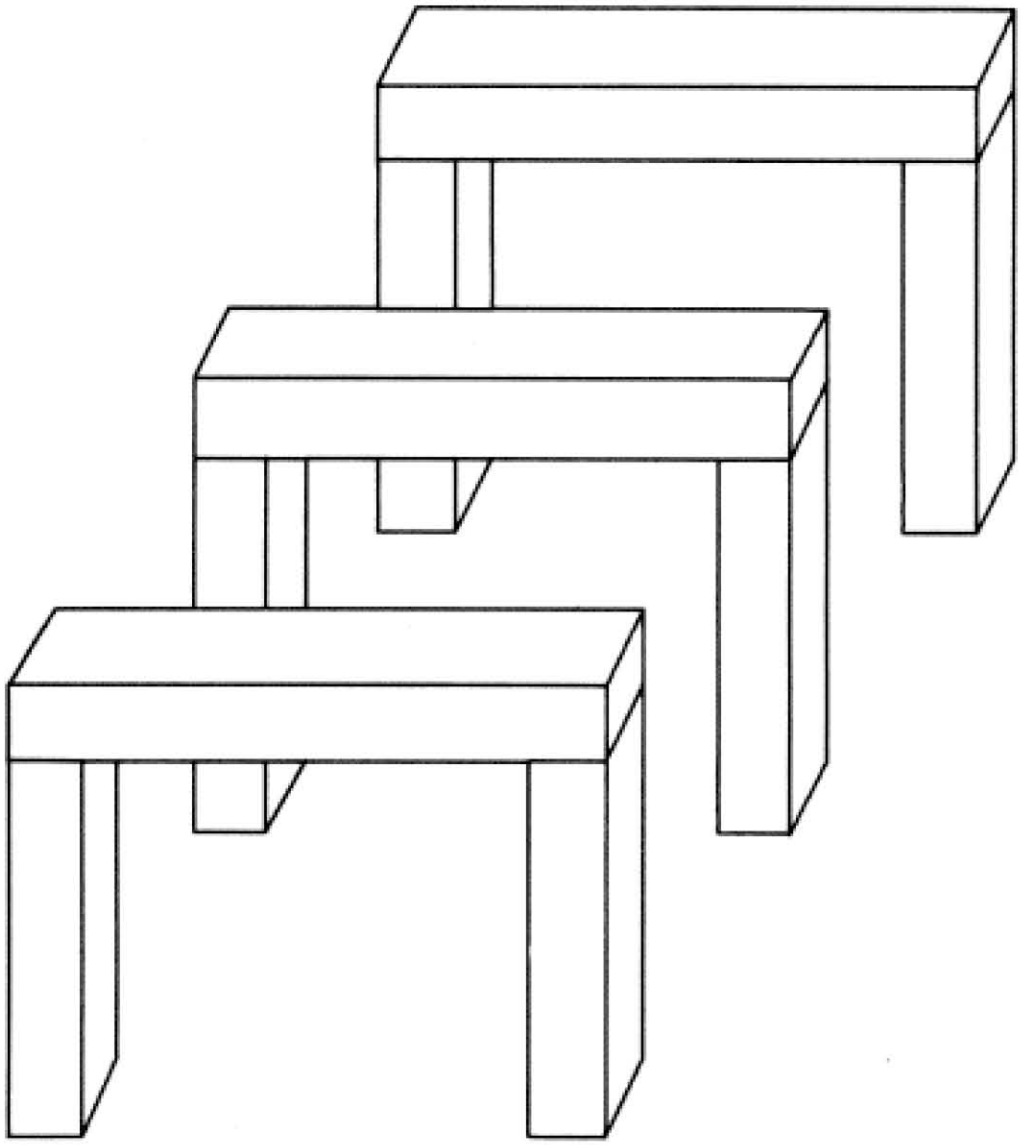


FIGURE 6-42: ARCADE

than of simply objects. As such the arcade development shows how the model builder can appeal to things already learned in the process of understanding more complex structures.

In building a description of an arcade, the description programs identify arches using the previously assimilated arch model. This leads to the description partially shown in figure 6-43. Then subsequent samples, shown in figure 6-44, figure 6-45, and figure 6-46 inform the machine that there must be a group, that the group elements must be arches, and that the relation must be IN-FRONT-OF.

> The deduction that one arch is in front of another involves methods less explored and less scund than techniques previously described for dealing with individual objects. Indeed this is a virgin field of inquiry that I have thought about only enough to write programs which can handle these few examples. It is not clear, for example, if each structure will require its own set of heuristics for determining inter-group relations, or if general principles can be enunciated. So far my primitive programs assume only the following rules:
>
> 1. If there is a chain of support relations between every object in structure A to some object in structure B, the B can be said to support A.
>
> 2. If some object in A is in front of some object in B, but not vice versa, then A can be said to be in front of B.

6.9 The Table

The table is much like previous examples except that grouping is done on the basis of object form and function, rather than relation chains.

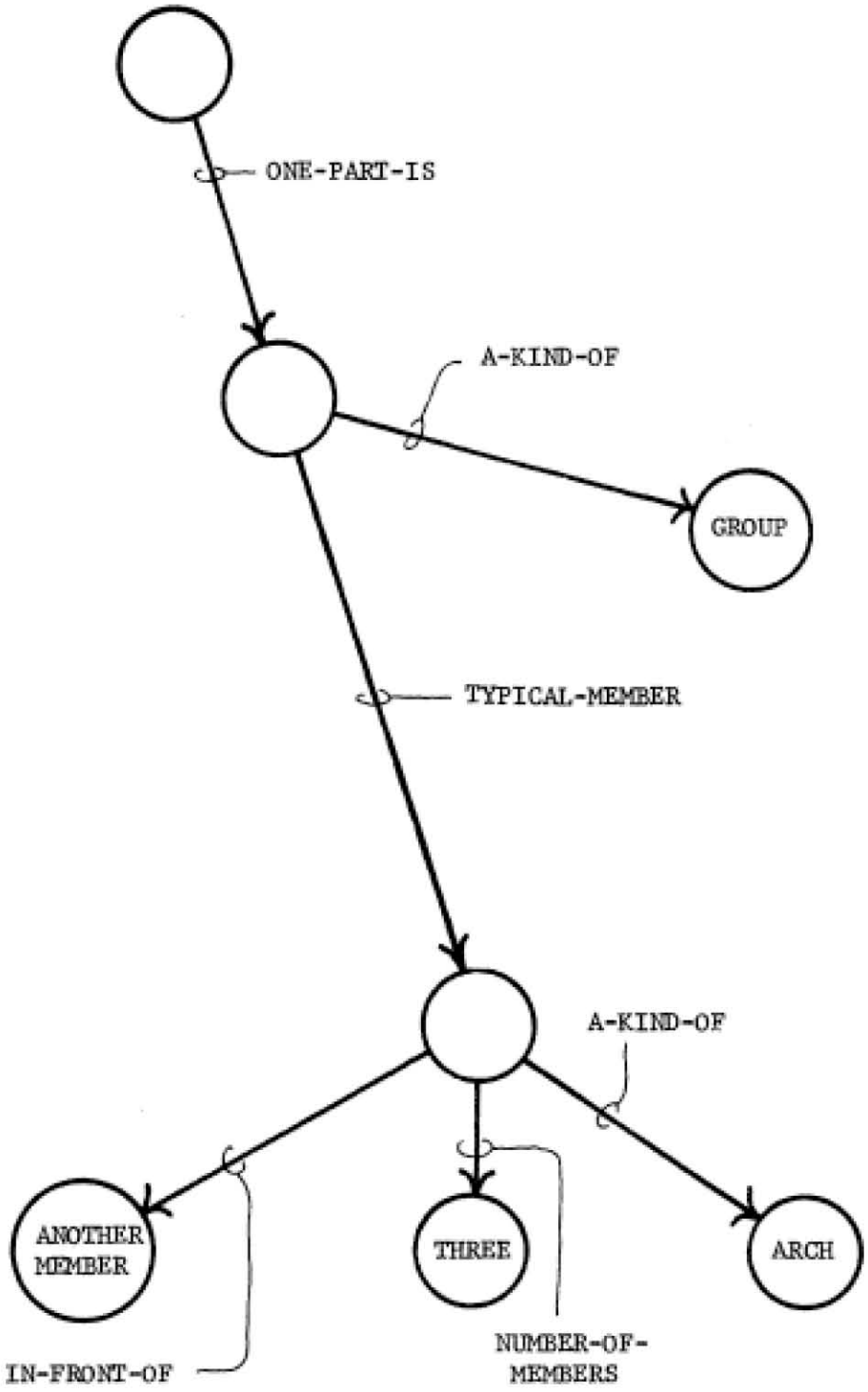

ONE-PART-IS
A-KIND-OF
GROUP
TYPICAL-MEMBER
A-KIND-OF
ANOTHER MEMBER
IN-FRONT-OF
THREE
NUMBER-OF-MEMBERS
ARCH

FIGURE 6-43

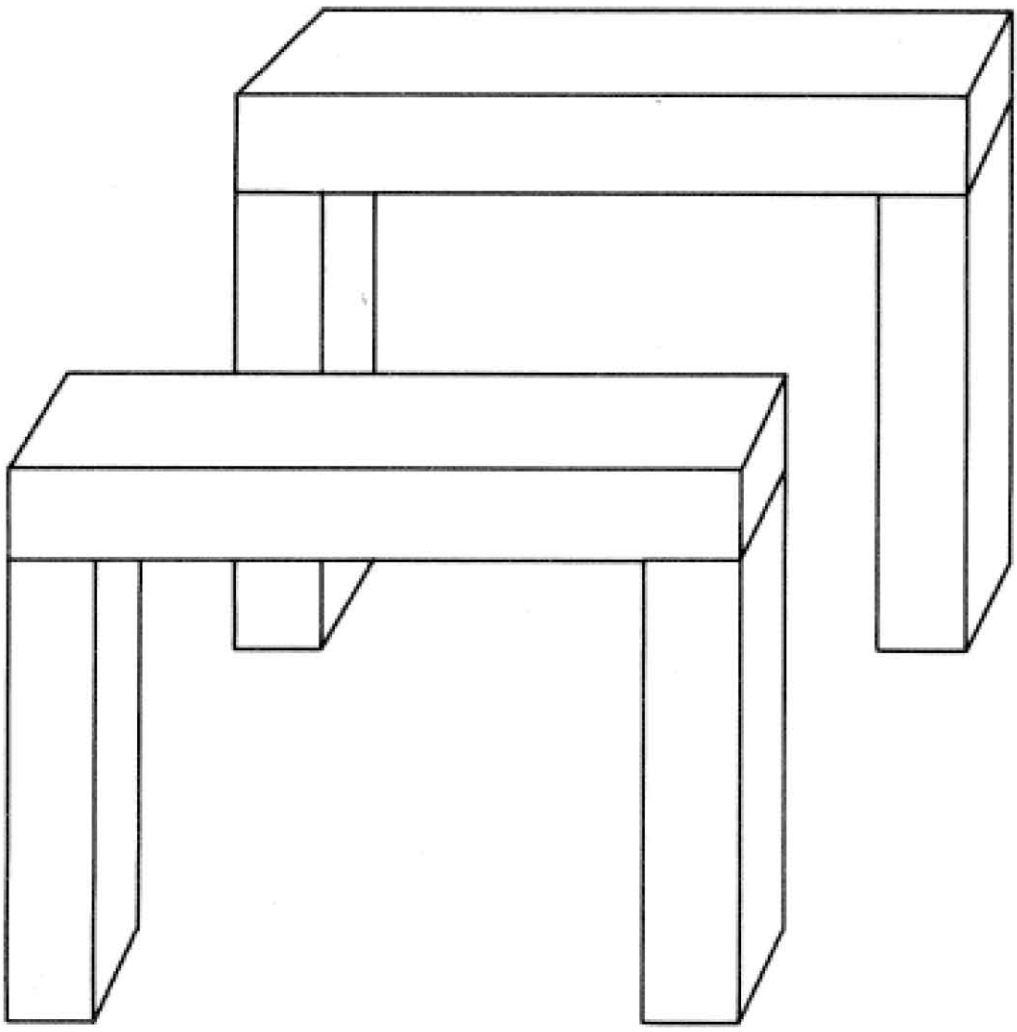


FIGURE 6-44: NEAR MISS TO ARCADE

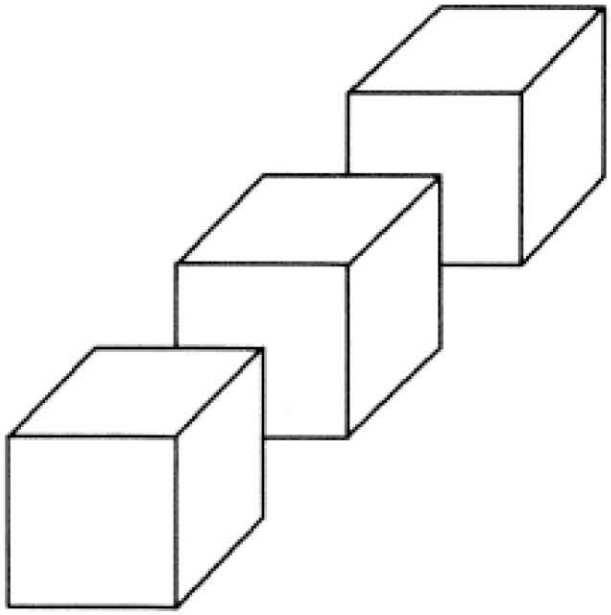


FIGURE 6-45: NEAR MISS TO ARCADE

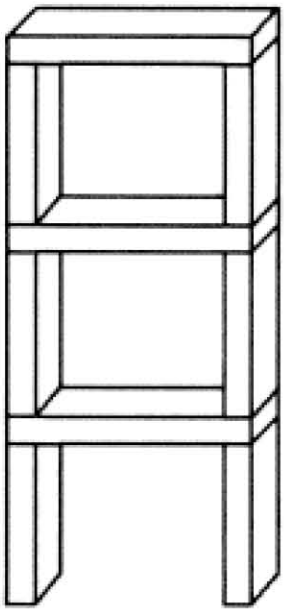


FIGURE 6-46: NEAR MISS TO ARCADE

Study the table in figure 6-47 and the description in figure 6-48. The essential features of the table are introduced by the following sequence of steps:

First the table should have bricks for legs. This idea is easily conveyed by the non-table of figure 6-49. Moreover, this conception of table excludes structures such as that in figure 6-50, a fact which is handily incorporated through a MUST-NOT-MARRY pointer. Next, since the non-table in figure 6-51 has only two supports, no grouping occurs, which leads to insistence on a group in the next model refinement. This entirely parallels the process by which the column was found to involve a group. Finally, the scene in figure 6-52 leads to replacement of the SUPPORTED-BY pointer by MUST-BE-SUPPORTED-BY. Figure 6-53 shows the last model in this development.

6.10 The Arch in Depth

So far the illustrations have shown networks only to that depth appropriate for understanding. Figure 6-54 shows the model for the arch in somewhat fuller bloom and better indicates the breadth of the information available to programs that use the model.

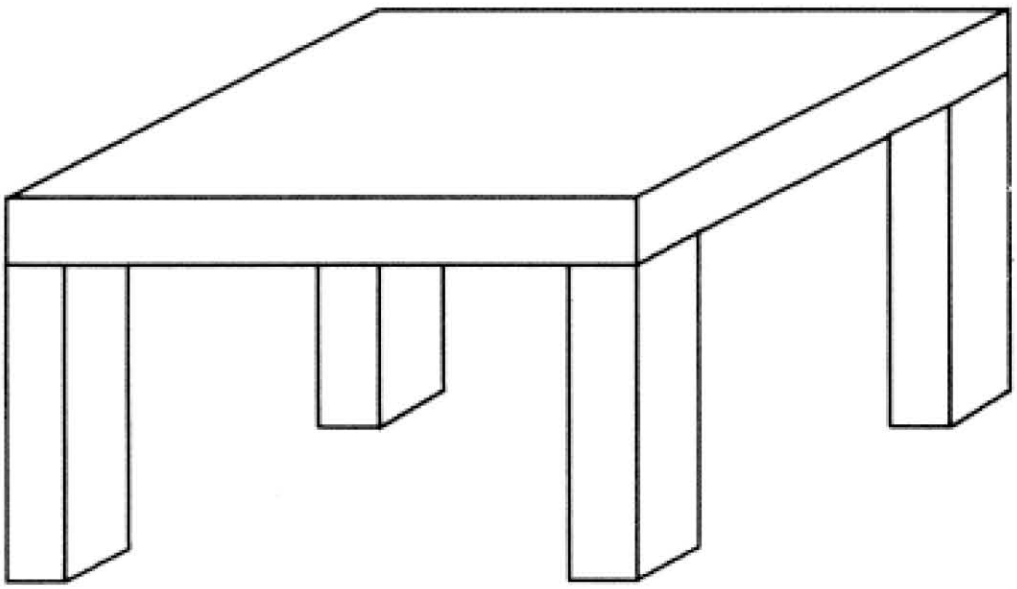


FIGURE 6-47: TABLE

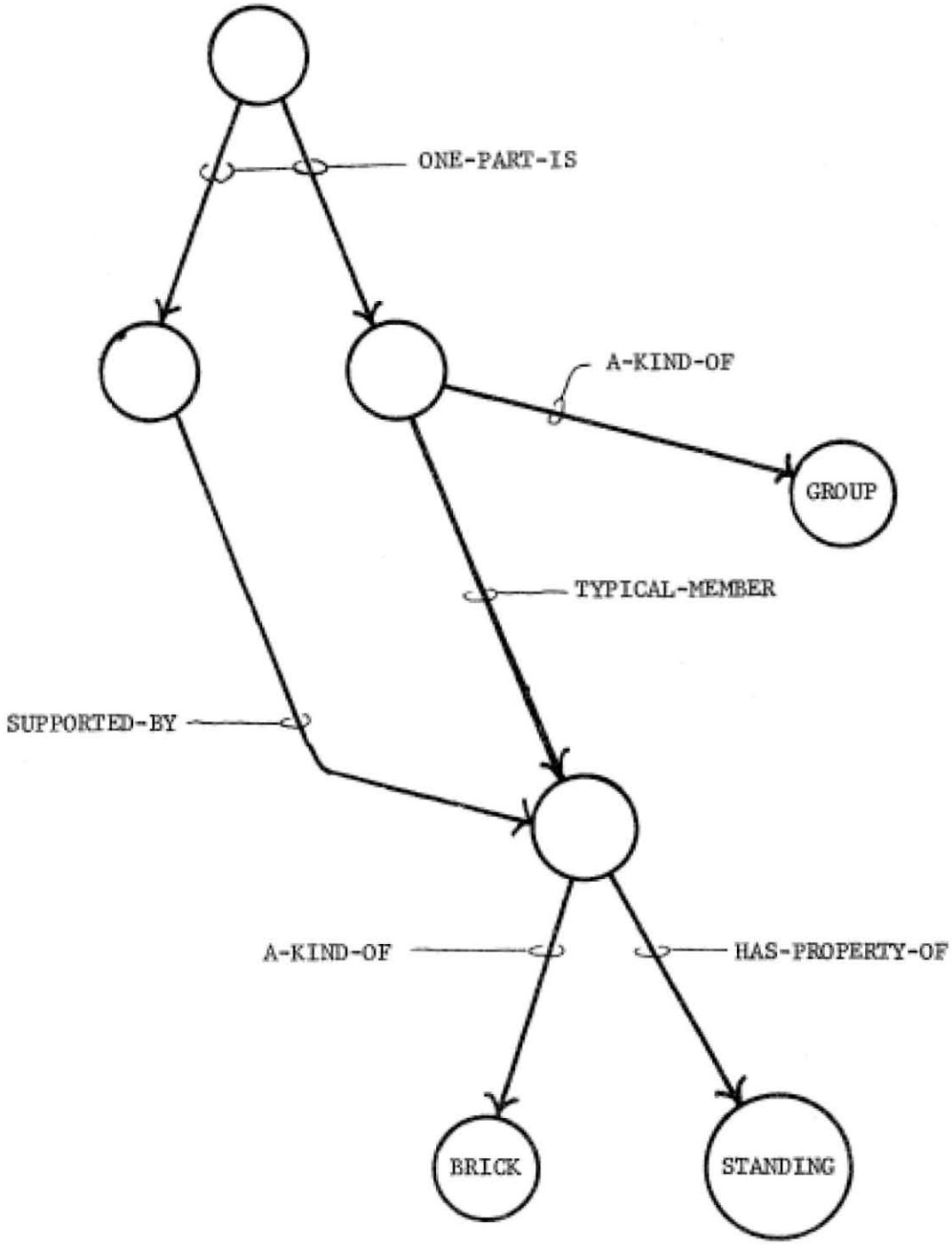

ONE-PART-IS
A-KIND-OF
GROUP
TYPICAL-MEMBER
SUPPORTED-BY
A-KIND-OF
HAS-PROPERTY-OF
BRICK
STANDING

FIGURE 6-48

TABLE

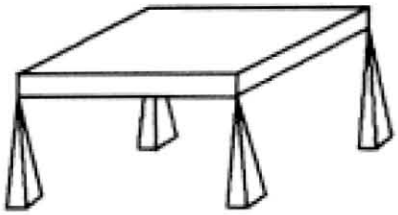


FIGURE 6-49

NEAR MISS

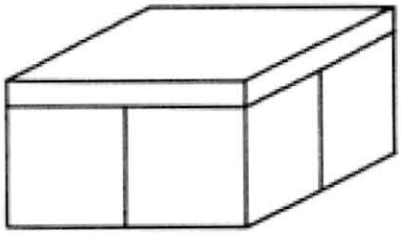


FIGURE 6-50

NEAR MISS

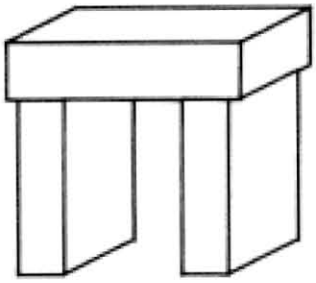


FIGURE 6-51

NEAR MISS

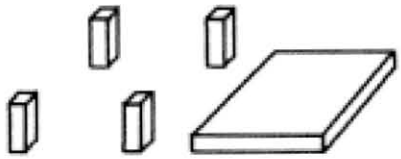


FIGURE 6-52

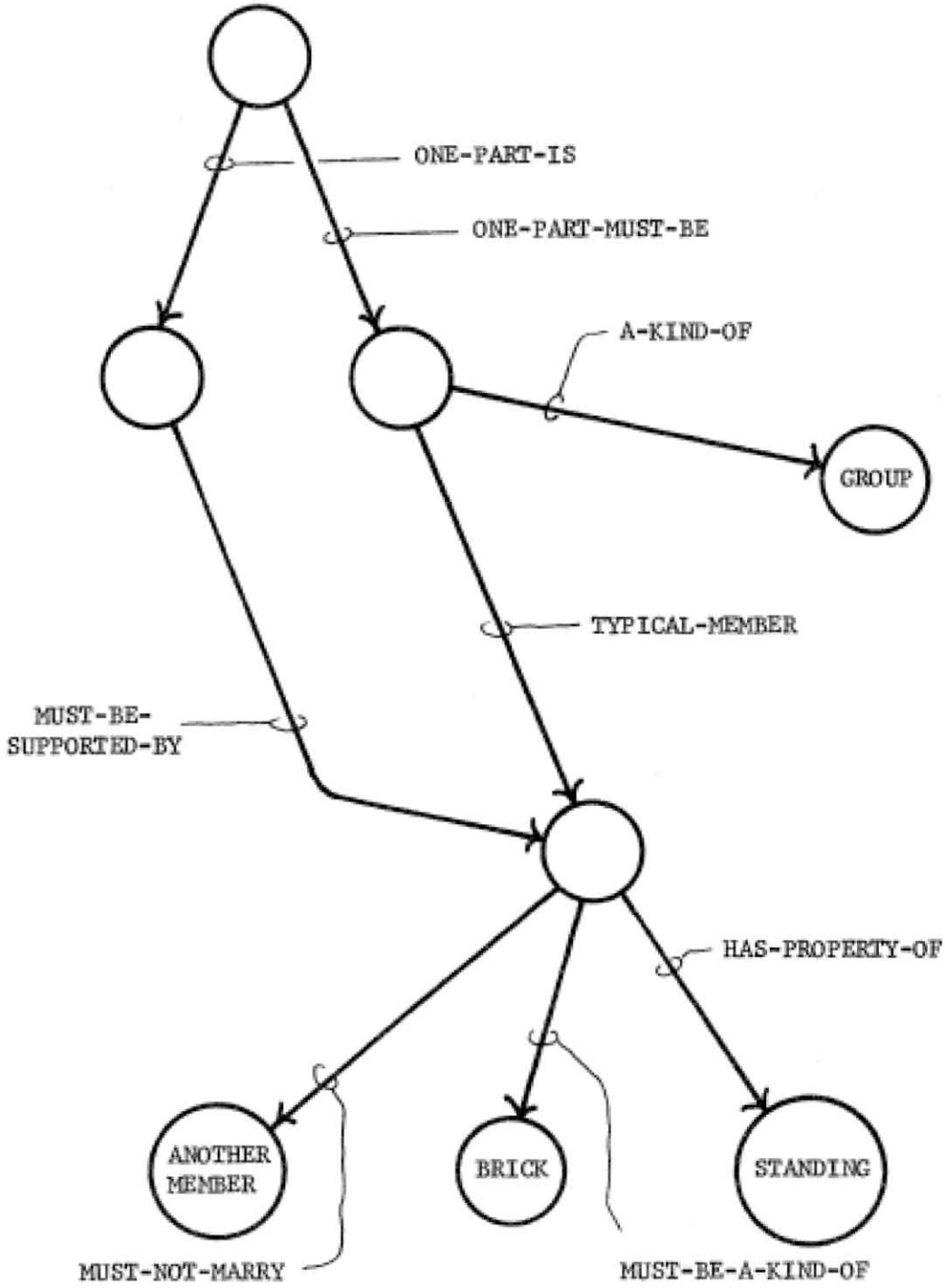

ONE-PART-IS
ONE-PART-MUST-BE
A-KIND-OF
GROUP
TYPICAL-MEMBER
MUST-BE-
SUPPORTED-BY
HAS-PROPERTY-OF
ANOTHER
MEMBER
BRICK
STANDING
MUST-NOT-MARRY
MUST-BE-A-KIND-OF

FIGURE 6-53

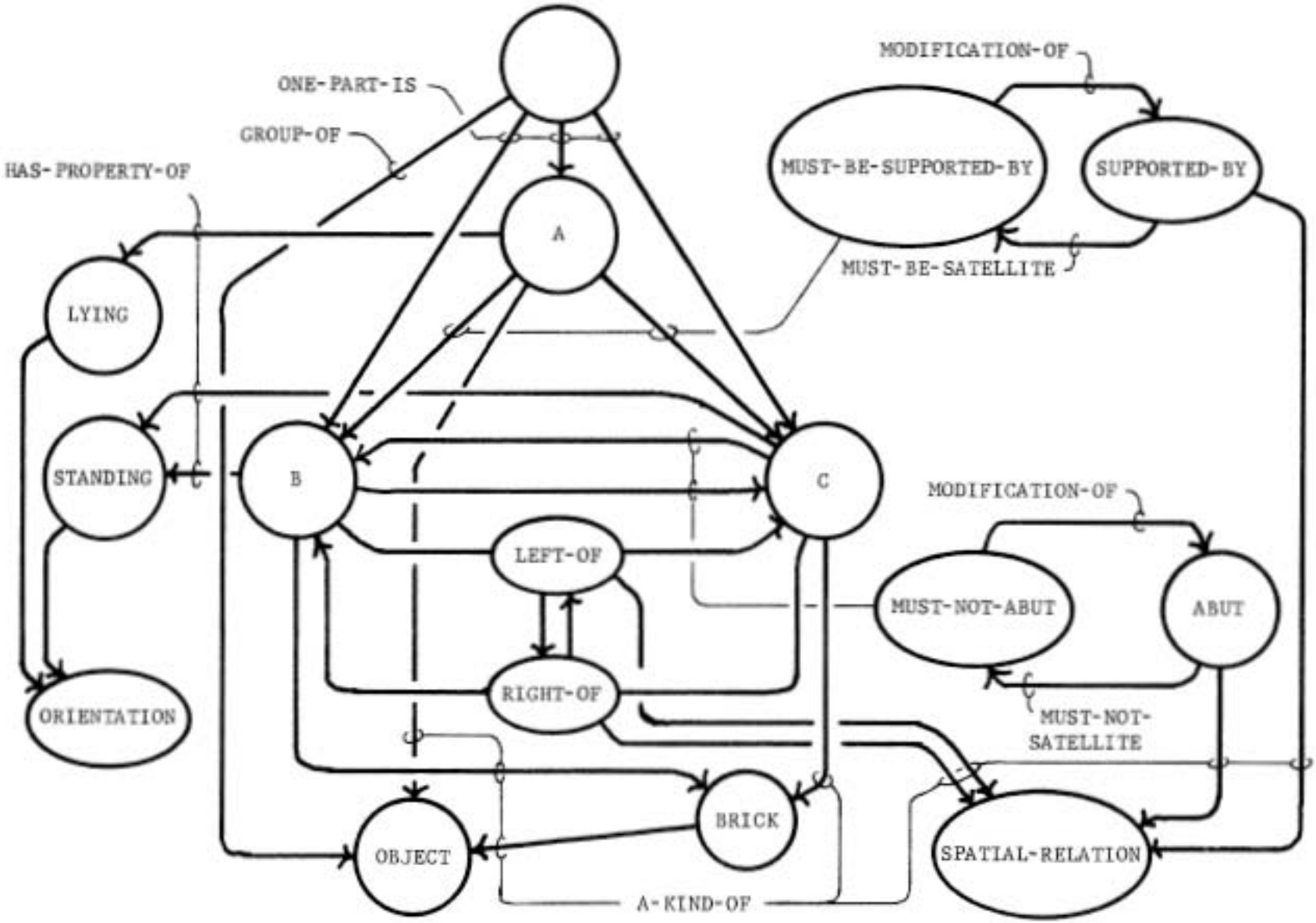

ONE-PART-IS
GROUP-OF
HAS-PROPERTY-OF
MODIFICATION-OF
MUST-BE-SUPPORTED-BY
SUPPORTED-BY
MUST-BE-SATELLITE
A
LYING
STANDING
B
C
LEFT-OF
MODIFICATION-OF
MUST-NOT-ABUT
ABUT
RIGHT-OF
ORIENTATION
MUST-NOT-
SATELLITE
BRICK
OBJECT
SPATIAL-RELATION
A-KIND-OF

FIGURE 6-54

6.11 Directions for improvement

Experiments with the concept generator titillate rather than satisfy, for each success suggests new ideas to be explored. As it stands, the concept generator is a healthy baby but not a contributing adult. The possibilities are staggering, however.

One way to improve model generation abilities is self-evident; the system improves as the ability to describe improves. The heuristic determination of support and occlusion and the rest can be improved, and more important, other so far neglected relations and properties lie unexplored. Description programs should and can be taught to differentiate between cubes, bricks, boards, and sheets. They should know in general when the terms lying and standing are meaningful. They should be able to relate structures more carefully. They should present alternative descriptions if situations are plainly ambiguous. They should know about color and texture. They should know about holes.

But as description becomes better, the burden on the model builder becomes greater, for the proliferation of properties and relations means that the number of c-notes multiplies, thereby complicating all of the decision processes. To cope, it will probably be necessary to institute further techniques for locating the centrally

important c-notes.

For one thing, c-notes involving certain pointers might rate priority attention. These might, for example, be pointers that frequently played central roles in the development of other models in the past. Additionally, some pointers might be tentatively converted to MUST-BE versions by virtue of frequent occurrence in the samples under consideration. Previous objections to this idea still stand, but done with care, with the assumptions behind the action somehow indicated, some good might come from this.

If the machine could ask the teacher questions, it would open up another powerful kind of finesse. In situations becoming precariously ambiguous, the model builder would ask directly if some relation is important, or perhaps display several imagined scenes to the teacher, requesting a statement about which are in the class to be learned.

Finally, and perhaps crucial, some confrontation needs to be made with function. What to do, however, is unclear. In dealing with the table, the development was already strained. It is not really adequate to think of legs solely as standing bricks, and generalization to the class of all objects seems specious. So far all models and identifications deal only with descriptions of concepts' parts, but this is not adequate to handle the notion of the

leg. What is needed is the idea that something is a leg, not because of what its parts are and how they are related, but rather because that something relates to something else in a particular way, namely through the SUPPORTED-BY relation. Given the ability to think this way, programs could identify round legs, square legs, or legs made of several parts. Functional use would serve as a very powerful interpretive and grouping tool adding immeasurably to the limited understanding now attained.

> I emphasize that no fundamental barrier prevents programs from thinking functionally. Indeed some possibly useful programs already exist. As work progresses, further analysis can be attempted and identification can be expanded to include the relationships that suggest function. At first work should concentrate on things that are defined in terms of currently observed use, rather than things that are defined in terms of conjectured potential use. A table leg, for example, is a leg because it currently is observed to support something. The same object would be far less likely to be identified by a human as a leg were it seen separated from any table. There is little possibility of identifying something as a table leg on the basis of potential use unless a leg is specifically searched for.

7 Identification

7.1 Matching and Identification Alternatives

Once there are programs that describe scenes, compare description networks, and build models, one may go on to using these programs as elements in a variety of other goal-oriented programs. The problem-solving programs described in this chapter have the following kind of responsibilities:

To see if two scenes are identical.

To contrast two scenes and report the differences between them, roughly in order of importance. This supplies information that may prove useful to programs that use a mechanical hand and arm to build copies of scenes.

To compare some scene with a list of models and report the most acceptable match. This is the identification problem in its simplest form.

To identify some particular subset of the objects in a scene. This is not the same as identifying an entire scene because important properties may be hidden and because context may make some identifications more probable than others.

To find instances of some particular model in a scene. It is frequently the case that the presence of some configuration can be confirmed even though it would not be found in the ordinary course of scene description. This requires the ability to discern groups with the required properties in spite of a shroud of irrelevant and distracting information. It is not unlike the problem of finding the bunny on a Playboy cover.

7.2 Exact Match

If two scenes are identical, then the networks describing those scenes must be isomorphic. The nodes of the two networks must relate with each other in the same ways, and the nodes must relate to general concepts such as BRICK and STANDING in the same ways. Consequently, comparing two such networks produces a simple kind of comparison description. There is a skeleton, which indicates how the parts of the scenes interrelate, and there is a group of intersection c-notes that describe how the parts of the scene are anchored to the general store of concepts. None of the other types of c-notes appear because identical scenes cannot produce two networks with the necessary aberrations of form.

Conversely, if comparison of two networks results in intersection c-notes only, then the parent scenes must be identical in the sense that the description generating mechanisms employed produce exactly matching networks. There can be variation, but nothing so great as to vary the action of the description generator. The scenes in figure 7-1 are identical with respect to the descriptive power of my programs because in both cases the relations observed are LEFT-OF and RIGHT-OF. More capable programs might complain that FAR-TO-THE-LEFT-OF and FAR-TO-THE-RIGHT-OF hold in one scene, while only LEFT-OF and RIGHT-OF hold in the other.

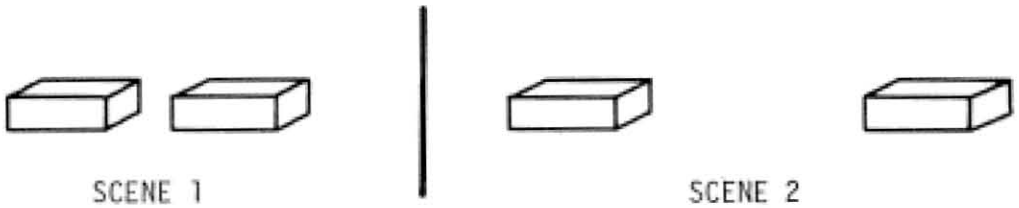


FIGURE 7-1

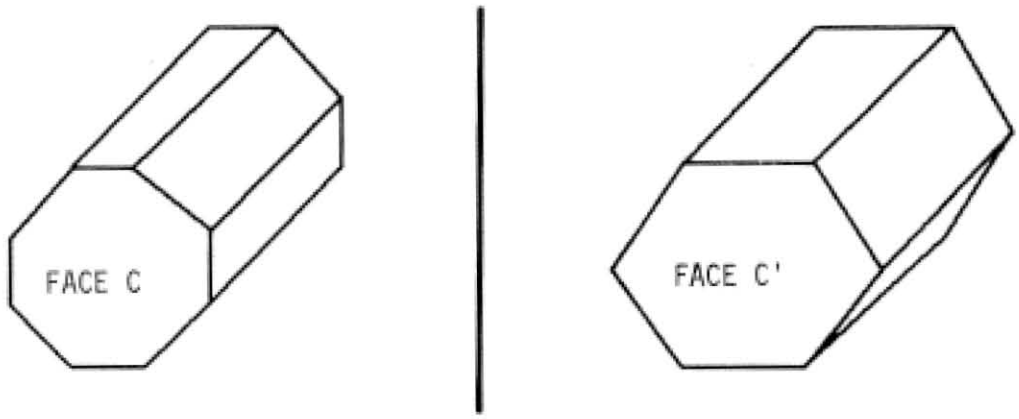


FIGURE 7-2

The scenes are clearly not identical with respect to a program with such a capability.

One generalization adds a degree of flexibility to this procedure that humans seem to exercise. The kind of question to be answered is not of the simple form "Are two given scenes identical?" but rather, "Are two given scenes identical with respect to a certain degree of detail?"

An approach to this new problem is to modify the intersection-only criterion. Instead of requiring all c-notes to be intersection c-notes, one requires that all c-notes be intersection c-notes at or above a certain level. Thus if level one is specified, then the scenes in figure 7-2 are indeed identical because the comparison description shows nothing but intersection c-notes at or above that level. (figure 7-3) But if level two is also of concern, then the scenes are not identical because an a-kind-of-merge c-note describing the difference in shape between face C and face C' appears on that level.

This use of level plainly defines a concrete substitute for the otherwise vague notion of degree of detail. One simply says two scenes either are or are not identical down to some particular level and the existence of non-intersection type c-notes beyond that level is not of concern.

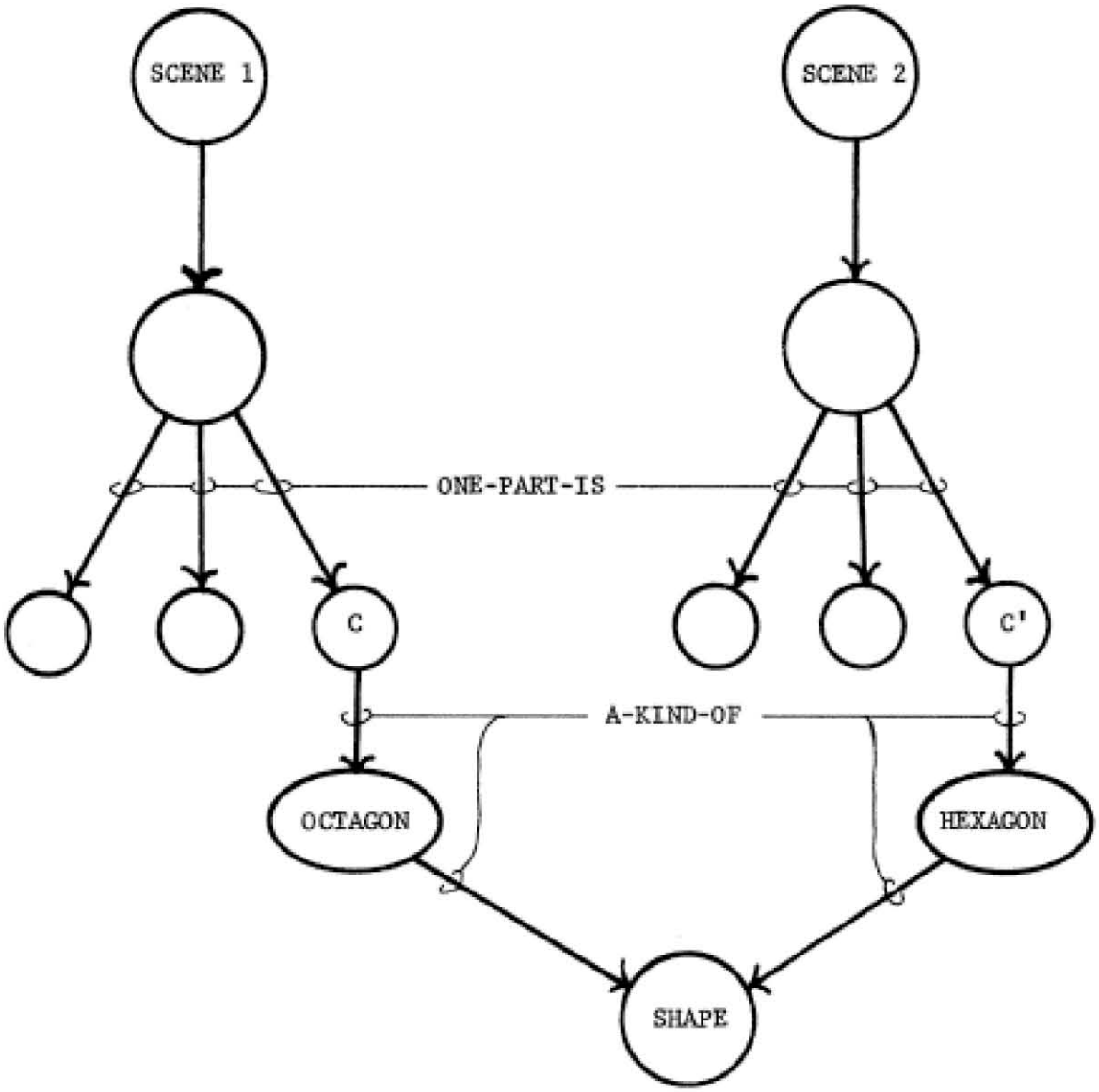


FIGURE 7-3

7.3 Digression

The exact match detector is a major part of a curiously simple program that checks for a certain kind of left-right symmetry. The method is as follows:

1. Copy the description of the scene exactly.
2. Convert all LEFT-OF pointers in the copy to RIGHT-OF, and all RIGHT-OF pointers to LEFT-OF.
3. Compare the original description against the modified copy. If the match is exact, the scene is symmetric.

This is, of course, an abstraction of the familiar condition for y-axis symmetry in the mathematical sense, whereby symmetry is confirmed if and only if for every point in the scene, (x,y), the point (-x,y) is also in the scene. Switching LEFT-OF and RIGHT-OF pointers is the analogue of x-coordinate negation and network matching corresponds to a check for invariance.

To see how this works, consider the scene in figure 7-4. The center object, A, is flanked by B on the left and by C on the right. Figure 7-5 shows the resulting description. There are nodes corresponding to objects A, B, and C, and there are LEFT-OF and RIGHT-OF pointers indicating their relationships.

Figure 7-6 shows the copy of the network with the LEFT-OF and RIGHT-OF pointers switched. Notice that the original network and the copy are identical. Node A matches with B',

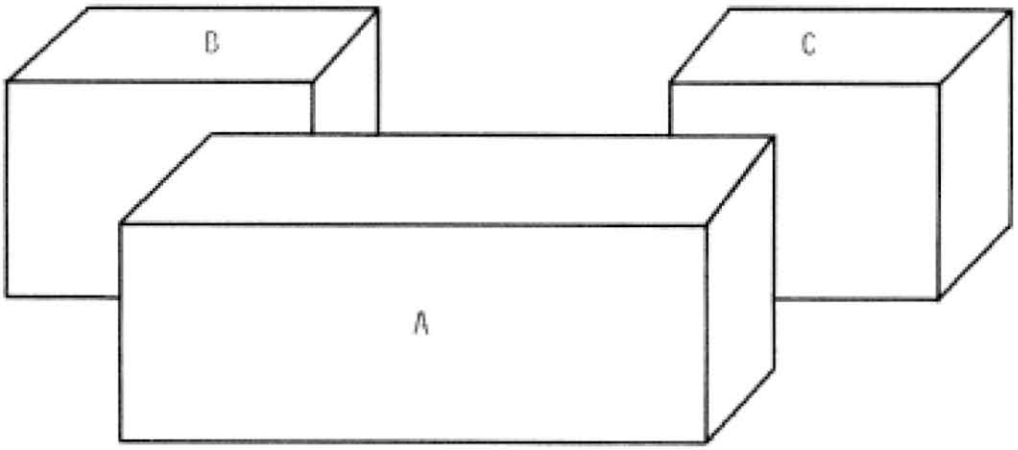


FIGURE 7-4

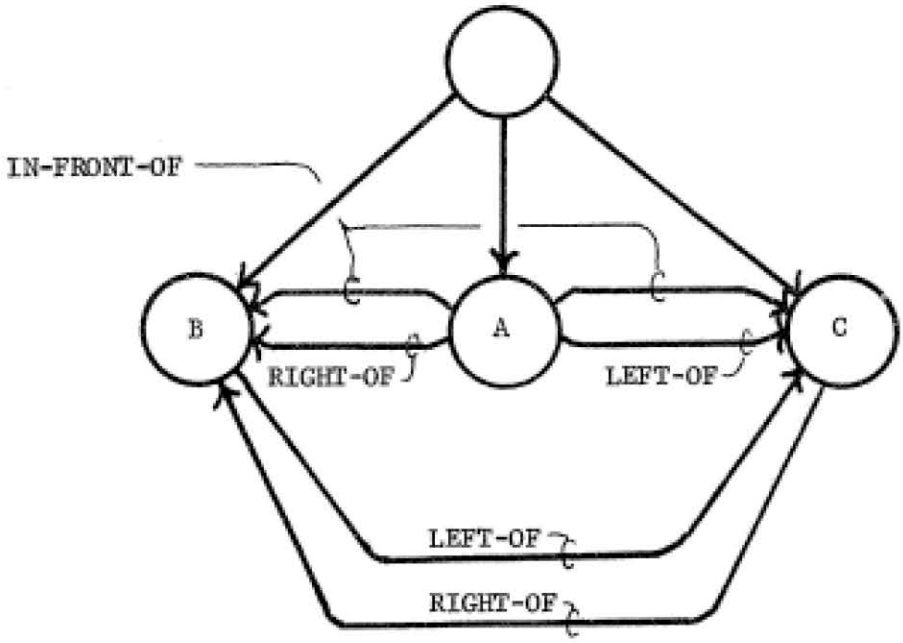


FIGURE 7-5

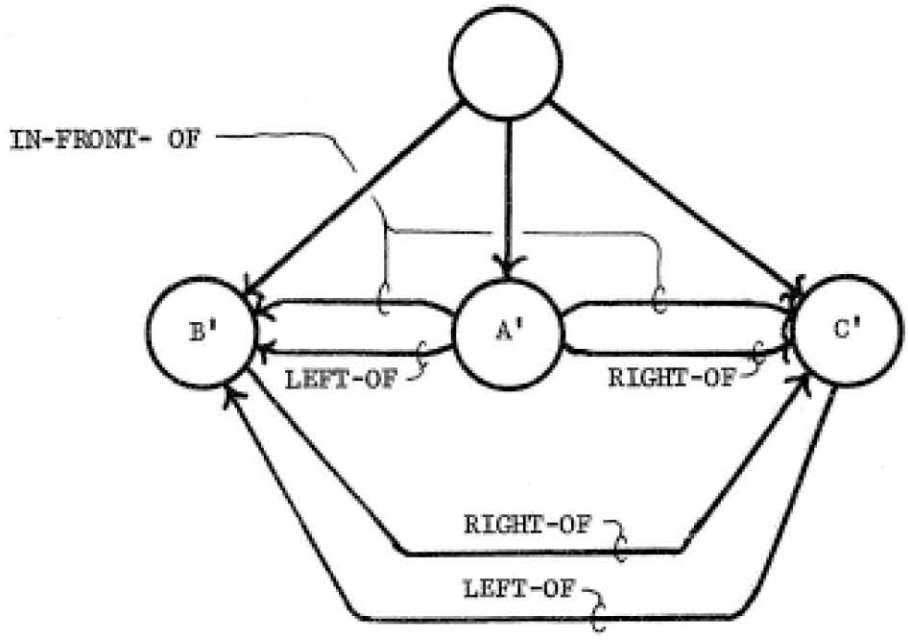


FIGURE 7-6

C with C', and B with A'. Since there are no differences, the machine concludes the scene is in fact symmetric.

Of course, a generalization of symmetry is possible, just as generalization of identity is. The machine need only perform the same copying and exchange operations and then check for non-intersection c-notes only to some particular depth. Thus the scene in figure 7-7 is symmetric to depth one because the groups of objects are symmetrically placed. It is not symmetric to depth two, however, because the placement of objects within the groups is wrong. The scene in figure 7-8 differs from that in figure 7-7 because there is not only symmetry in the location of groups of objects, there is also symmetry in the placement of objects within the groups. This means that the scene is more symmetric to the machine in the sense that the symmetry detection program remains happy at a deeper level of inquiry.

The machine knows LEFT-OF and RIGHT-OF are opposites because information about these pointers lies in the general memory net. (figure 7-9) Consequently, it is unnecessary to tell the program explicitly to substitute RIGHT-OF for LEFT-OF and vice-versa. One need only ask the symmetry program if there is symmetry with respect to either the pointer LEFT-OF or RIGHT-OF. The machine itself can conjure up the appropriate substitutions by working through the OPPOSITE

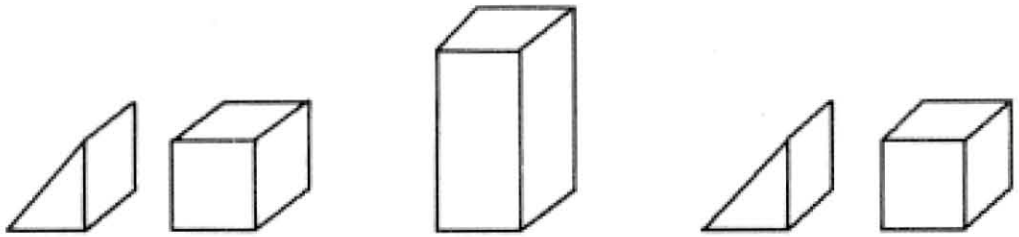


FIGURE 7-7

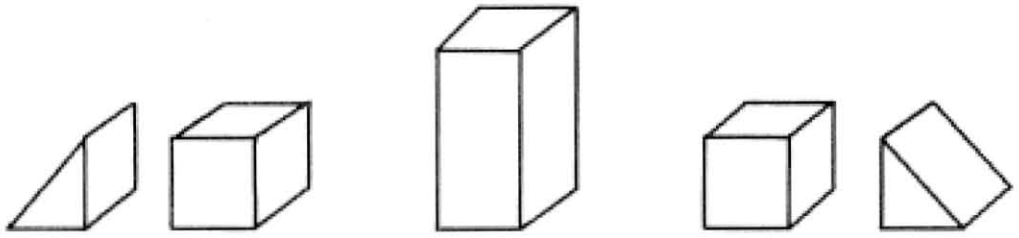


FIGURE 7-8

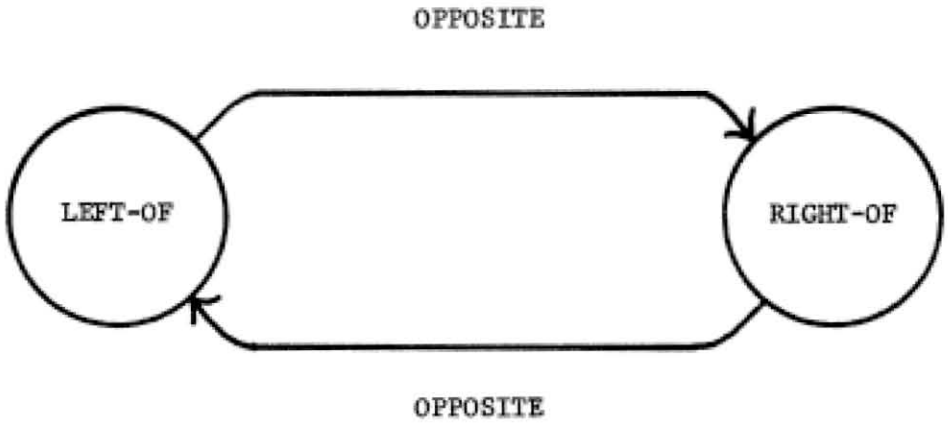


FIGURE 7-9

pointer from whichever relation is supplied, be it LEFT-OF or RIGHT-OF. Similarly, if one asks for symmetry with respect to ABOVE, the program realizes that the proper substitutions are BELOW for ABOVE and ABOVE for BELOW. IN-FRONT-OF gives BEHIND for IN-FRONT-OF and vice-versa for a somewhat unusual kind of symmetry question.

Certainly mixtures are also possible. One can ask for both left-right and above-below substitutions which is an abstraction of symmetry with respect to a point.

> Mathematically speaking, there is symmetry with respect to the particular point (0,0) if for every point (x,y) in the scene, there is also a point (-x,-y). The way to check a drawing is to imagine moving every point straight through the origin until it is again the same distance from the origin but lies in the opposite quadrant. For example, point P in the symmetric drawing in figure 7-10 goes to point P'. When the end result is the same as the original drawing, then the drawing is symmetric.

More important is a combination of a left-right and an in-front-of -- behind switch. This one gives the machine a chance of realizing that two scenes are just front and back views of the same configuration as are the scenes in figure 7-11.

> Eventually I think the machine can come upon the symmetry notion in the same way it now learns about arches and houses. But at this point I do not think there is enough comparison describing capability. The needed step is the introduction of a program that generates global c-notes from the local ones already at hand, thereby introducing the kind of hierarchy into the comparison descriptions that is already the standard in

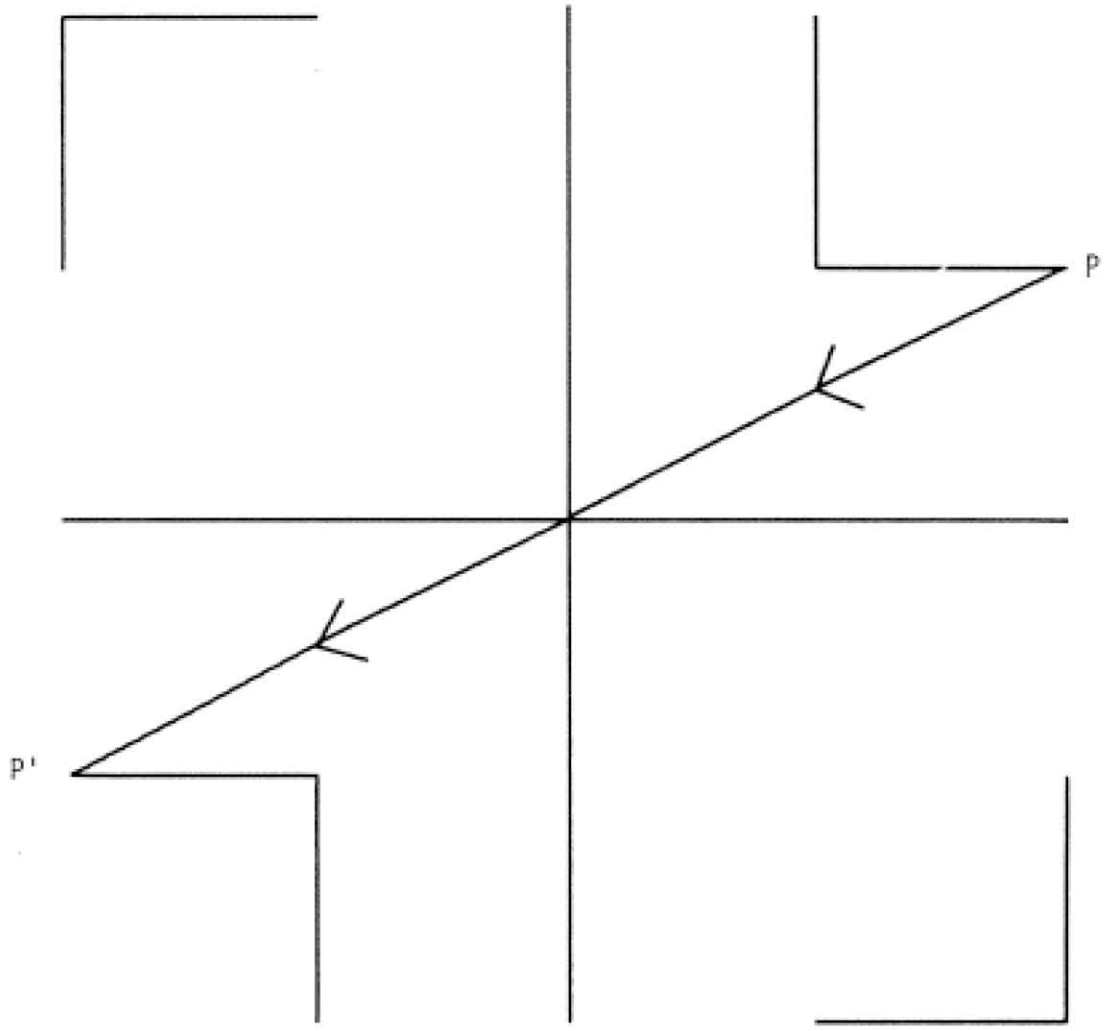


FIGURE 7-10

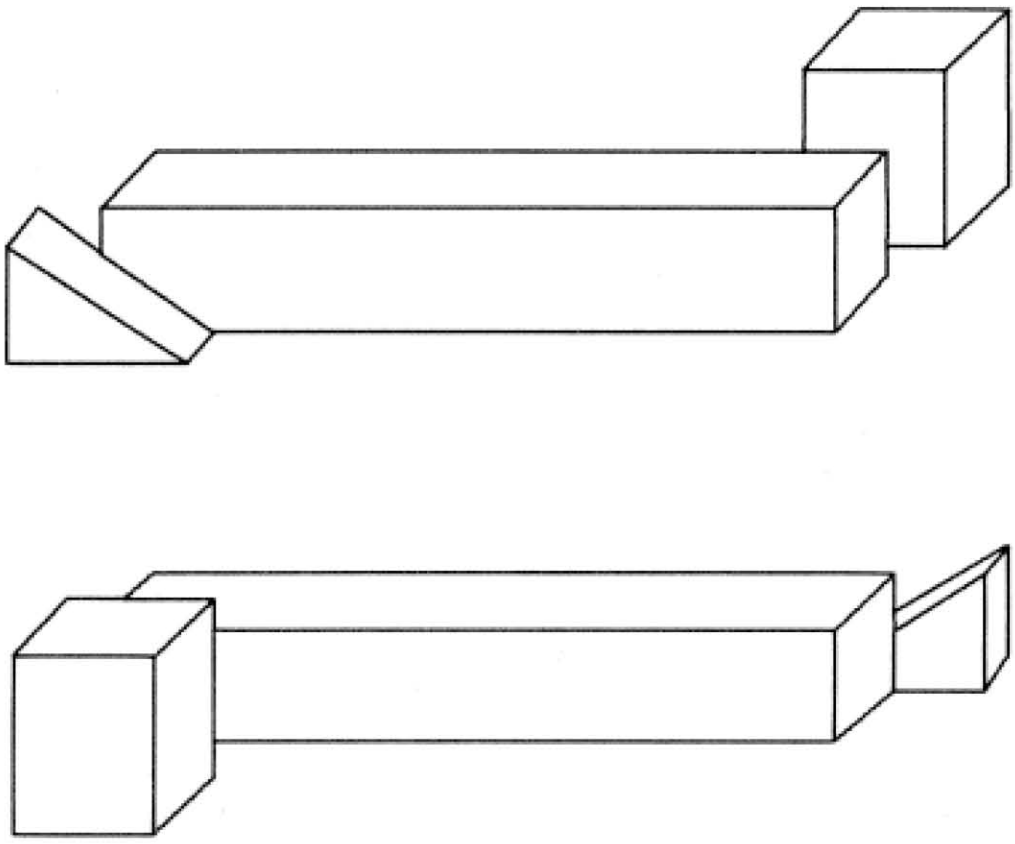


FIGURE 7-11

scene descriptions. One obvious ability of such a program would be that of noticing a preponderance of similar c-notes. I think that this and some of the double comparison ideas proven useful in doing analogy problems are just the things the machine needs to learn about symmetry.

7.4 Elementary Identification

Suppose a scene is to be identified, if possible, as a HOUSE, PEDESTAL, TENT, or ARCH. The obvious procedure is to match its description against those for each of the models and then somehow determine which of the four resulting difference descriptions implies the best match.

Recall that models generally contain must-be satellites and must-not-be satellites while ordinary descriptions do not. Consequently, comparing an ordinary description against a model leads to a variety of c-notes not found when ordinary descriptions are compared. Among these are must-be-satellite pairs, must-not-be-satellite pairs, and various flavors of exits and supplementary-pointers. Such c-notes are decisive in the identification process.

Consider the case where some pointer in a scene's description corresponds to its must-not-be satellite in the model. This clearly means a relation is present that the model specifically forbids. The resulting must-not-be-satellite-pair c-note in the difference network is such a serious association impediment that identification of the

unknown with the model is rejected outright, without further consideration. This means that the near-arch in figure 7-12 cannot be identified as an arch because the network describing the near-arch has MARRYS pointers between the two supports while the model has MUST-NOT-MARRY pointers in the same place. The combination produces a comparison description with a must-not-be-satellite-pair c-note that positively prevents match.

Identification with a particular model is also rejected if the difference description contains exits or supplementary-pointer c-notes which involve must-be satellites. Such c-notes occur when essential relations or properties are missing in the unknown. Thus the two bricks in figure 7-13 do not form a pedestal because the model for the pedestal has a MUST-BE-SUPPORTED-BY pointer where the scene of figure 7-13 has nothing. The result is a supplementary-pointer c-note involving the must-be satellite MUST-BE-SUPPORTED-BY. Again match fails.

The case of a-kind-of-merge c-notes involves a slightly more complicated rule. Recall that merge c-notes occur generally when two nodes share properties that are not identical, but which fall into the same general class. The situation must be one where two linked nodes exhibit closely related pointers to two other nodes from which paths of A-

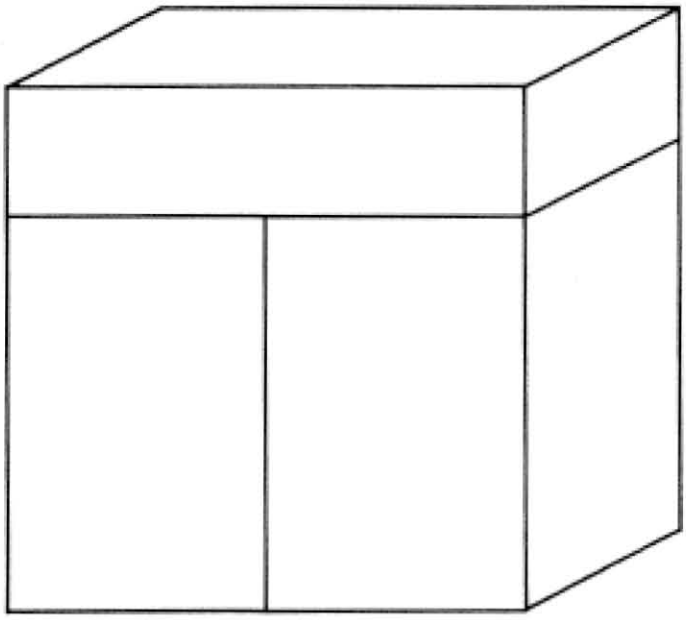
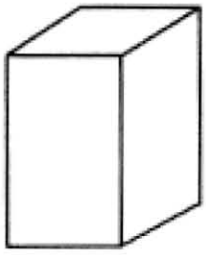


FIGURE 7-12



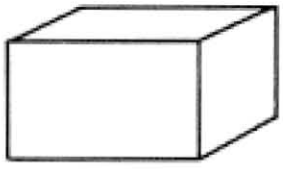


FIGURE 7-13

KIND-OF pointers lead to a common intersection. Figure 7-14 shows such a situation. In this case the unknown is a kind of wedge while the corresponding object in the model must be a kind of brick. Both WEDGE and BRICK are kinds of objects, which directly leads to a merge c-note associated with a MUST-BE-A-KIND-OF pointer in the model. But the fact that the unknown has a property in the same class as a property required by the model is insufficient. To insure rejection of such matches, the rule is: Refuse identification if the model's pointer contributing to the merge c-note is a must-be-satellite.

Figure 7-15 summarizes the procedure used on each c-note.

Match of the scene in figure 7-16 against the PEDESTAL, the TENT, and the ARCH all lead to difference descriptions with c-notes that forbid identification. The PEDESTAL fails because a merge indicates that the required A-KIND-OF relation between the top object and BRICK is missing. The TENT similarly fails because both of its objects must be wedges. The ARCH fails because the model has a MUST-BE-SUPPORTED-BY pointer to an object missing in the unknown. This in turn causes a fatal exit c-note in the difference description.

Of course the machine can also match the sample

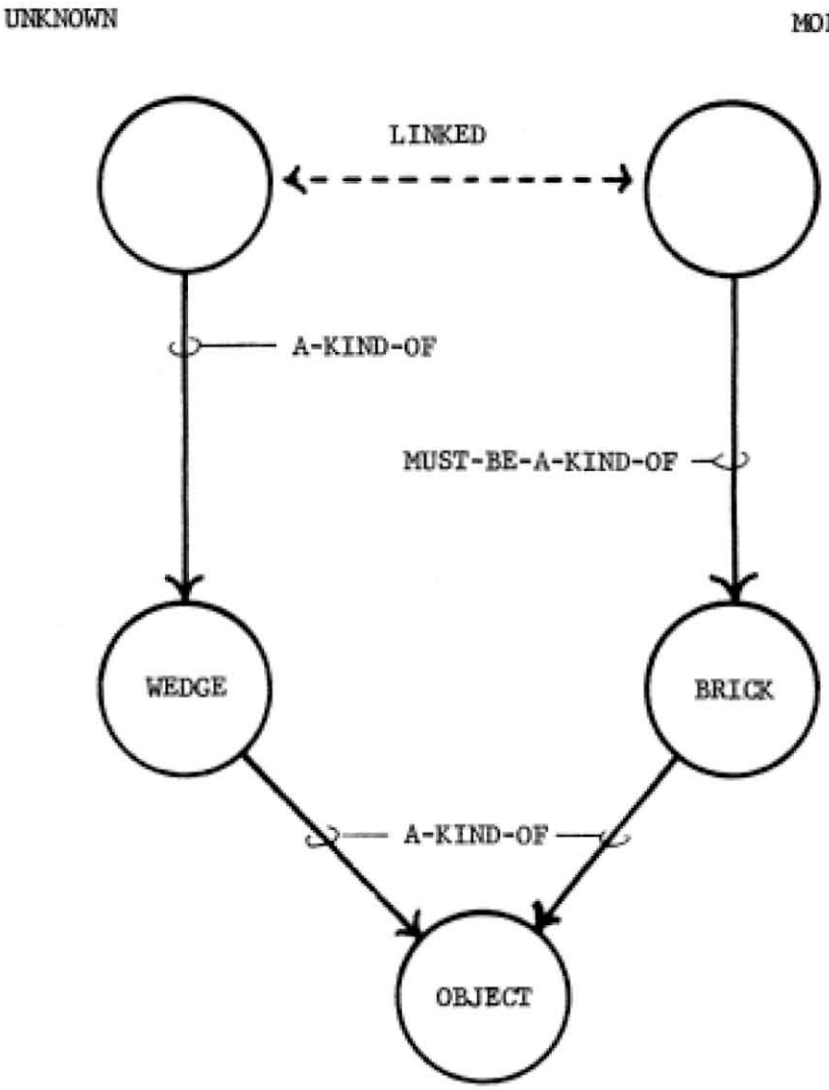

UNKNOWN
MODEL
LINKED
A-KIND-OF
MUST-BE-A-KIND-OF
WEDGE
BRICK
A-KIND-OF
OBJECT

FIGURE 7-14

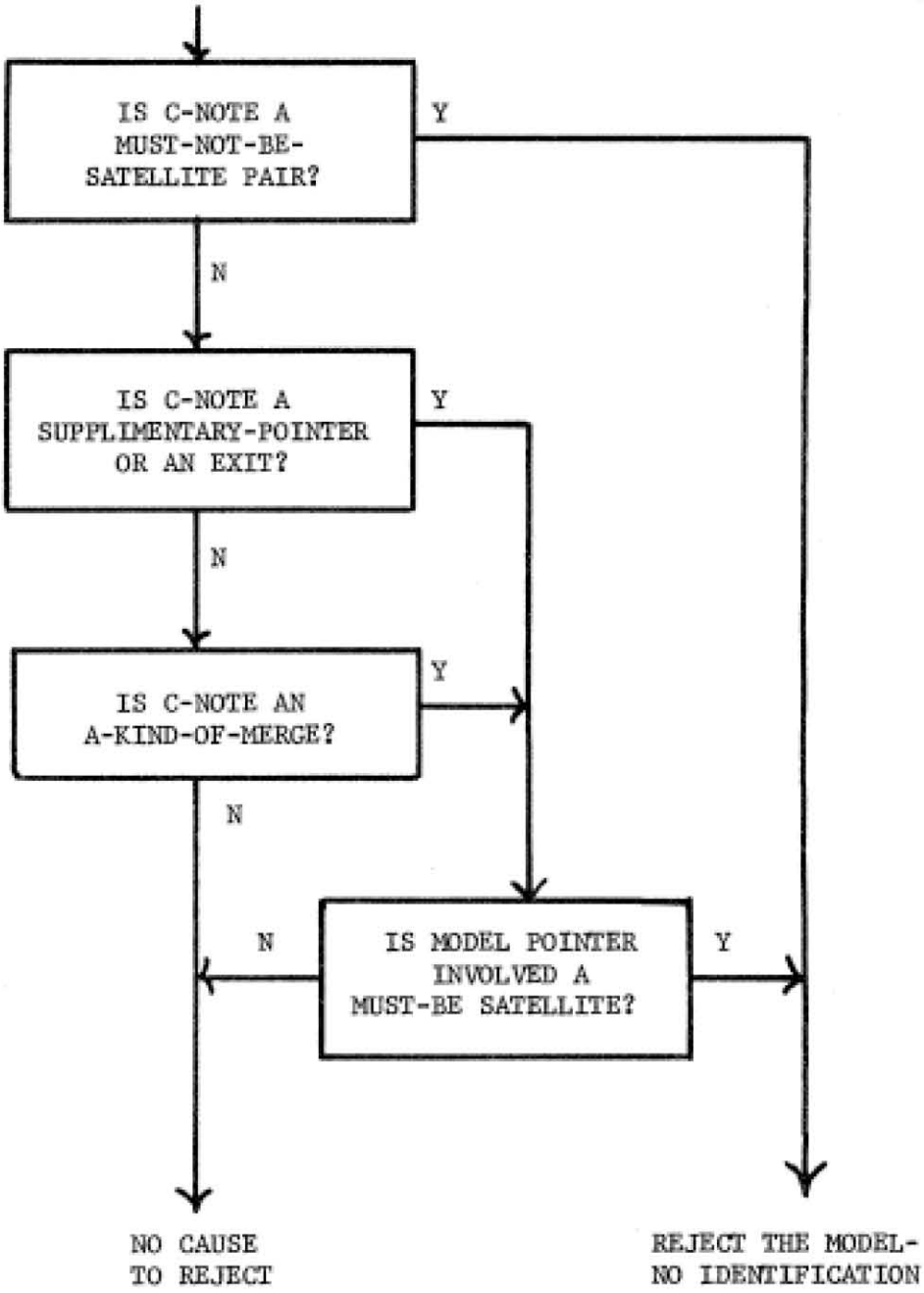

IS C-NOTE A MUST-NOT-BE-SATELLITE PAIR?
Y
N
IS C-NOTE A SUPPLIMENTARY-POINTER OR AN EXIT?
Y
N
IS C-NOTE AN A-KIND-OF-MERGE?
Y
N
IS MODEL POINTER INVOLVED A MUST-BE SATELLITE?
N
Y
NO CAUSE TO REJECT
REJECT THE MODEL-NO IDENTIFICATION

FIGURE 7-15

HOUSE

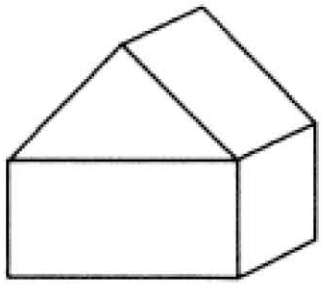


FIGURE 7-16

PEDESTAL

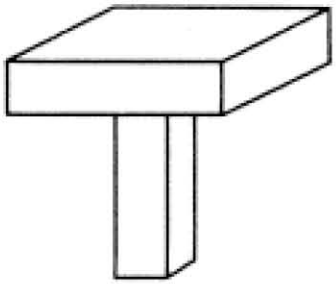


FIGURE 7-17

TENT

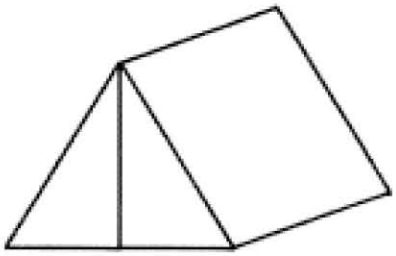


FIGURE 7-18

ARCH

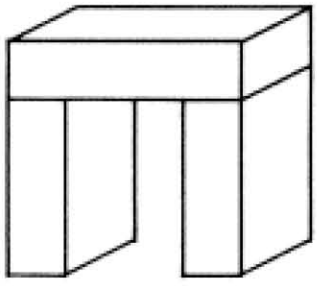


FIGURE 7-19

pedestal, tent, and arch of figure 7-17, figure 7-18, and figure 7-19 against the same list of models. It makes the correct identification in each case.

The next problem emerges because some unknown may acceptably match more than one model in a trial list. Suppose one defines a new sort of arch that is just like the old arch except that the top object must be a wedge rather than just any object. Call this new model the WEDGE-ARCH. Then the scene in figure 7-20 certainly matches with both the ORDINARY-ARCH and the new WEDGE-ARCH. There is only one slight variation in the difference descriptions. In the WEDGE-ARCH case, one has a must-be-satellite-pair c-note because the unknown has an A-KIND-OF pointer to WEDGE and the model has a MUST-BE-A-KIND-OF pointer. In the ORDINARY-ARCH case, there is simply an A-KIND-OF pointer from the model to OBJECT, which with the unknown's A-KIND-OF pointer to WEDGE forms an a-kind-of-merge c-note.

Of course there is nothing really wrong with reporting both ORDINARY-ARCH and WEDGE-ARCH as the identification of the unknown. Still, given several possible identifications, there should be some way of ordering them such that one could be reported to be best in some sense. To do this I associate each kind of difference with a number and combine the numbers to form a score for each comparison. Figure 7-21 shows the

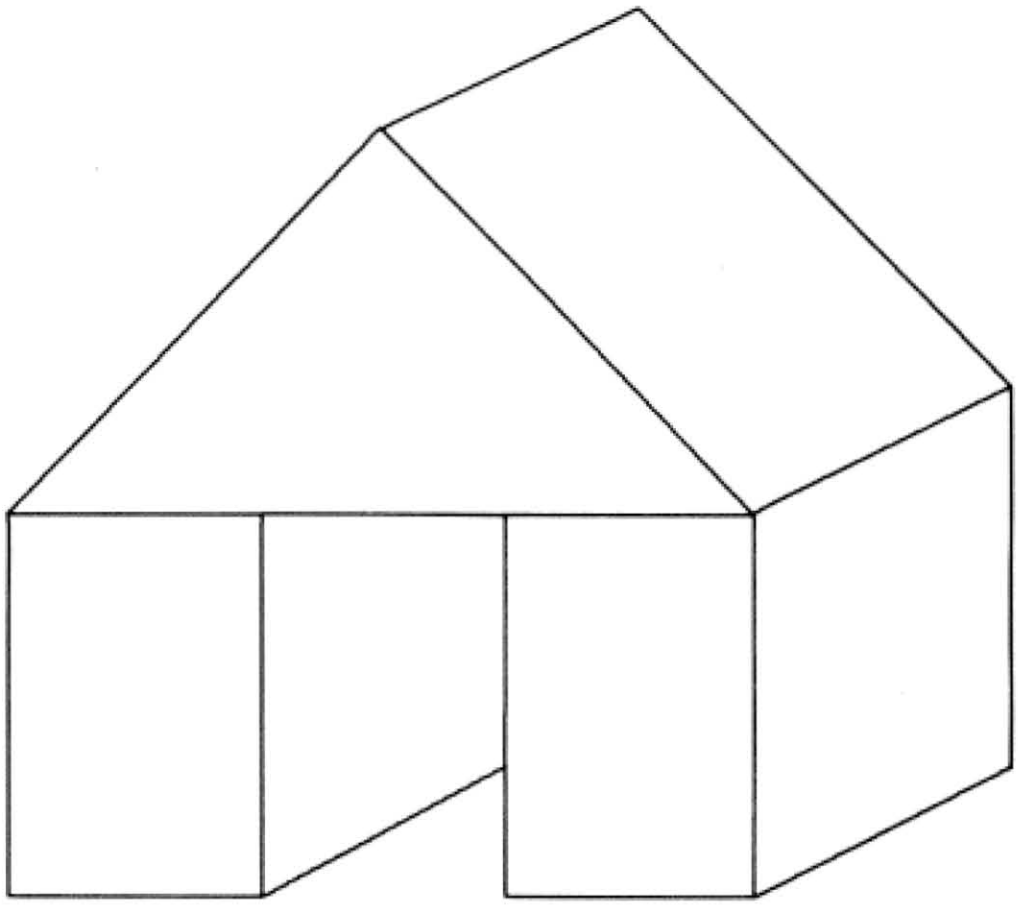


FIGURE 7-20

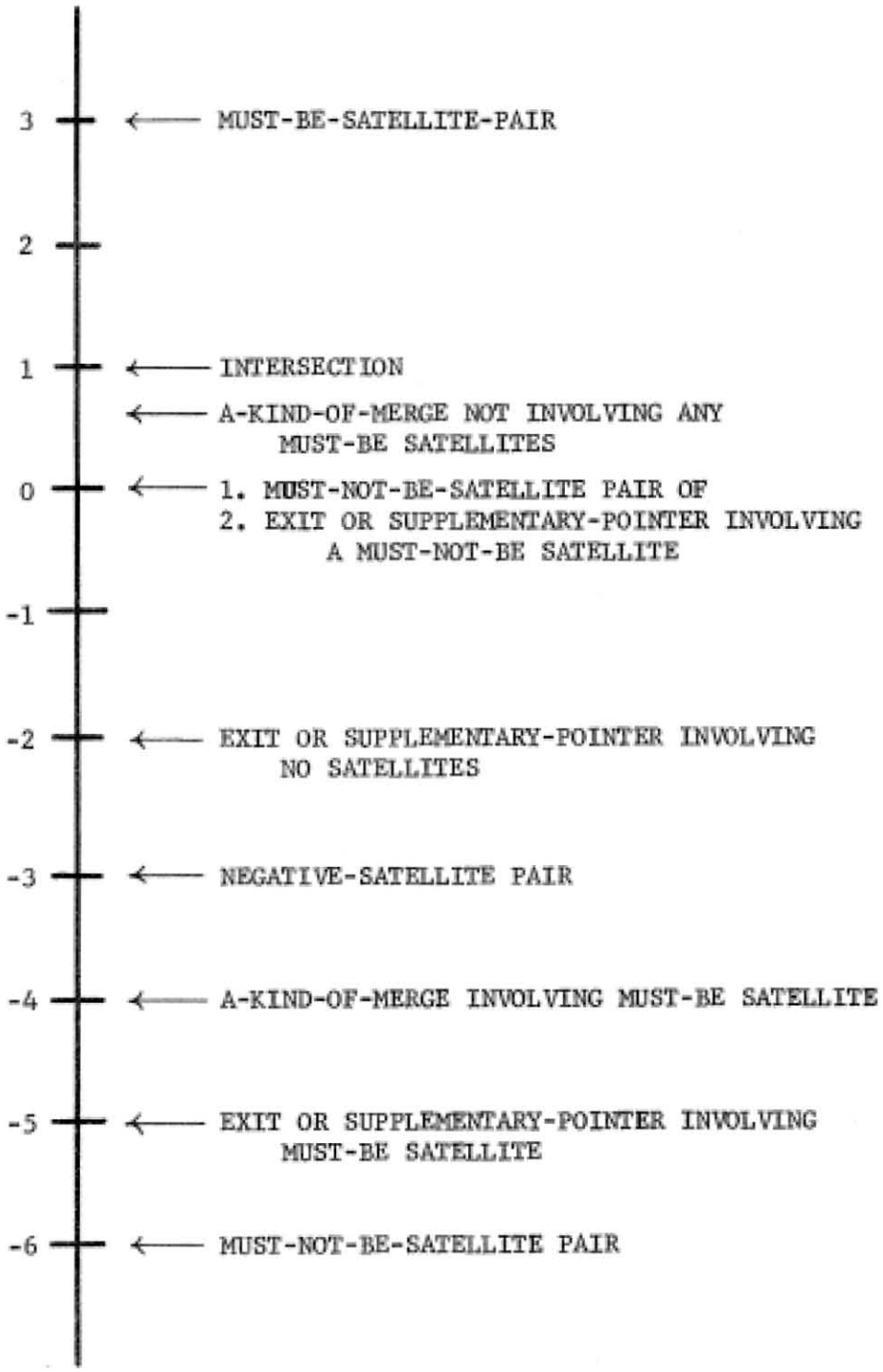

3
MUST-BE-SATELLITE-PAIR
2
1
INTERSECTION
A-KIND-OF-MERGE NOT INVOLVING ANY MUST-BE SATELLITES
0
1. MUST-NOT-BE-SATELLITE PAIR OF
2. EXIT OR SUPPLEMENTARY-POINTER INVOLVING A MUST-NOT-BE SATELLITE
-1
-2
EXIT OR SUPPLEMENTARY-POINTER INVOLVING NO SATELLITES
-3
NEGATIVE-SATELLITE PAIR
-4
A-KIND-OF-MERGE INVOLVING MUST-BE SATELLITE
-5
EXIT OR SUPPLEMENTARY-POINTER INVOLVING MUST-BE SATELLITE
-6
MUST-NOT-BE-SATELLITE PAIR

FIGURE 7-21

scale associating difference types with numbers. It evolves heuristically from observations like the following:

1. The intersection c-note has an assigned value of 1. This anchors the scale as all other numbers are fixed according to how good or bad the corresponding c-note seems relative to the intersection c-note.

2. A must-be-satellite-pair c-note suggests good match even more strongly than the intersection because it indicates that relations are present that are known to be essential. A value of 3 gives it three times the weight of a simple intersection.

3. Exit or supplementary-pointer c-notes that involve must-be-satellites are distinctively bad because they indicate vital properties are missing. The value is a damaging -5.

4. Other exit and supplementary-pointer c-notes are bad but not nearly so bad. A score of -2 seems about right.

5. Must-not-be-satellite pairs are very bad evidence indeed. The worst score of -6 is deserved.

6. The a-kind-of-merge is positive or negative depending on whether either of the pointers are must-be satellites. If a must-be satellite is involved, an important property is missing, resulting in a -4. Otherwise, it indicates loose association, not as tight as that announced by an intersection. A .5 is used.

Once differences are noted and number associations are made, a program must combine the numbers in a reasonable way. If SCORE[U:M] represents the score of comparing unknown U against model M, then I use

SCORE[U:M] = W(1)N(1) + +W(n)N(n)

where

W(i) = exp[-L(i)]

and

N(i) is the number associated with the ith difference.

W(i) is the weighting factor that reduces the influence of lower level differences.

L(i) is the level of the ith difference.

Combining the terms additively is convenient, and the weighting terms, the Ws, handily reduce the influence of the lower level differences. I have no stronger reasons for using this linear formula, and it is not something to be defended to the death. But I do not think it would pay to put much effort into tuning such a formula because more knowledge about the priorities of differences should lead to far better programs that do not use numbers at all.

7.5 Identification in Context

Examine figure 7-22. Notice that object B seems to be a brick while object D seems to be a wedge. This is curious because B and D show exactly the same arrangement of lines and faces. The result also seems at odds with the machine's models and identification process, as described so far, because so far anything identified as a

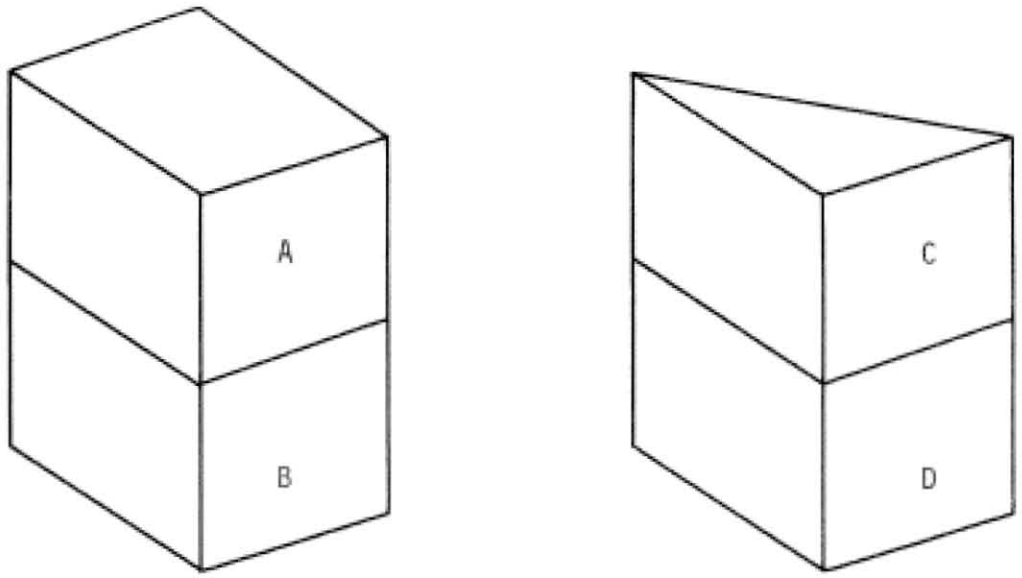


FIGURE 7-22

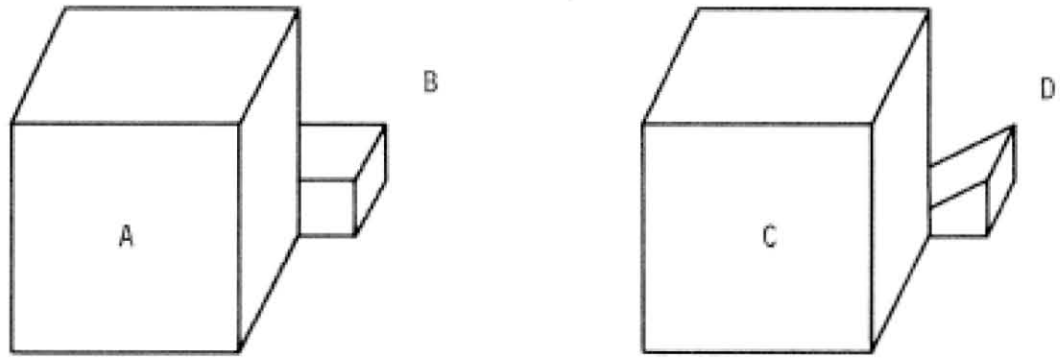


FIGURE 7-23

wedge must have a triangular face.

But of course context is the explanation. Different rules must be used when programs try to identify objects or groups of objects that are only parts of scenes, rather than the whole scene. In the case where the question is whether or not the whole scene can be identified as a particular model, it is reasonable to insist that all relations deemed essential by the model be present, while all those forbidden, be absent. But when the question is whether or not a few parts of a scene can be identified as a particular model, then there is the possibility that some important part may be obscured by other objects. In these situations, my identification program uses two special heuristics:

First, the coincidence of objects lying in a line seems to suggest that each object is the same type as the one obscuring it unless there is good reason to reject this hypothesis. This is what suggests object D is a wedge in figure 7-22.

Second, essential properties in the model may be absent in the unknown because the parts involved are hidden. This is why identification of object D with wedge works even though D lacks the otherwise essential triangular face. The requirement that forbidden

properties do not occur remains in force, however.

Elaborate work can be done on the problem of deciding if the omission of a particular feature of some model is admissable in any particular situation. My program takes a singularly crude view and ignores all omissions. Rejection of the hypothesis that the obscured is like the obscuror happens only if the machine notices details specifically forbidden by relations in the model. Thus the effort is not to select the test matching model, but only to verify that a particular identification is not contradictory. This means that object B in figure 7-23 is confirmed to be brick-like while brick-ness is denied to D because of the ruinous apparent triangularity of the side face.

Of course if the propagation of a property like brick-ness or wedge-ness down a series of objects is interrupted, then the unknown must be compared with a battery of models, with the program still forgiving omissions, but now searching for the best of many possible identifications.

No matter what the method by which a partly obscured object is identified, the use of a PROBABLY-A-KIND-OF pointer instead of the basic A-KIND-OF is used to qualify the conjectured relationship between the object and the

model it is identified with.

Figure 7-24 shows how the various pieces of this procedure fit together, and figure 7-25 shows what happens when it moves down a simple row of objects.

7.6 Learning from Mistakes

Suppose the program attempts to identify the scene in figure 7-26 as a pedestal. Identification fails because the resulting type of a-kind-of-merge c-note cannot be tolerated. Still it would be a pity to throw away the information about why the match failed. Instead the otherwise wasted matching effort can be used to suggest new identification candidates.

The way this works is quite simple. First the machine spends idle time comparing the various models in its armamentorium with each other. Whenever the number of differences observed are few, a simplified description of those differences is stored. Thus the machine knows that a house is similar to a pedestal, from which it differs only in the nature of the top object.

These descriptions link the known models together in a sort of similarity network. (figure 7-27)

This network and the difference descriptions noted in the course of identification failure help decide what model should be tried next. The description describing

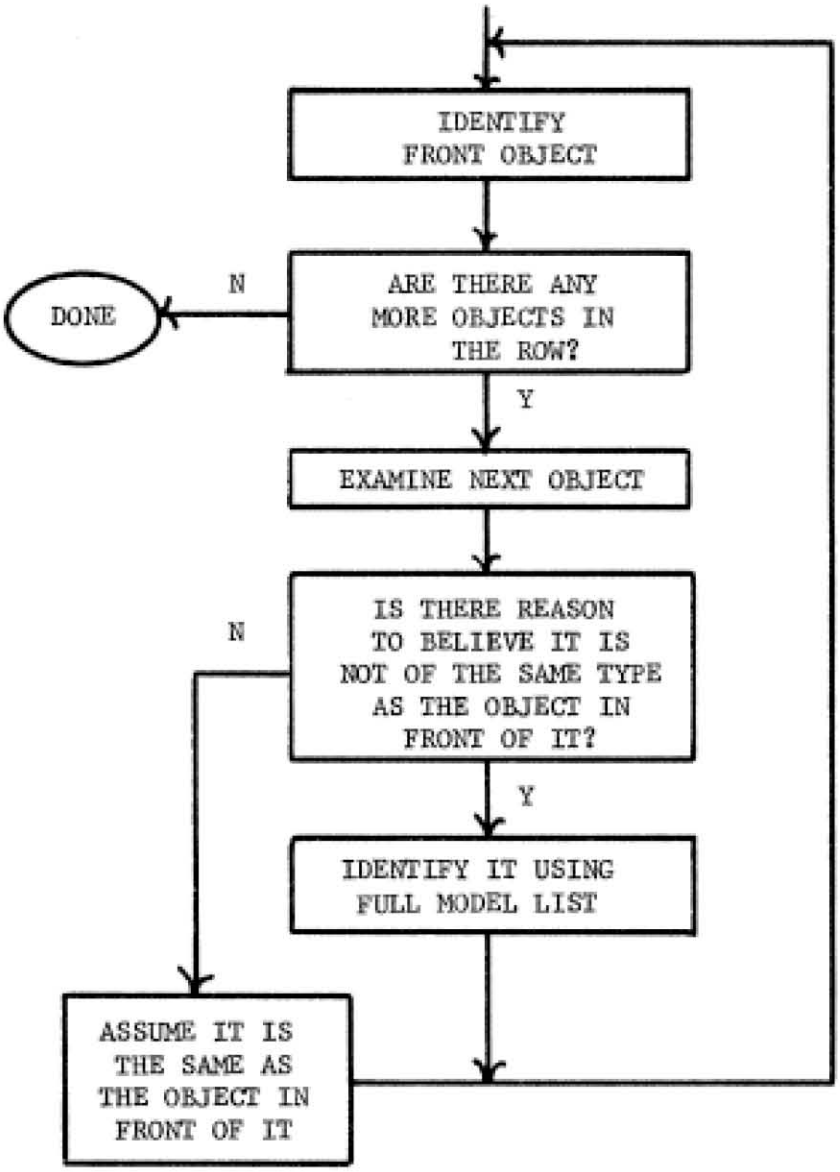

IDENTIFY FRONT OBJECT
ARE THERE ANY MORE OBJECTS IN THE ROW?
N
DONE
Y
EXAMINE NEXT OBJECT
IS THERE REASON TO BELIEVE IT IS NOT OF THE SAME TYPE AS THE OBJECT IN FRONT OF IT?
N
Y
IDENTIFY IT USING FULL MODEL LIST
ASSUME IT IS THE SAME AS THE OBJECT IN FRONT OF IT

FIGURE 7-24

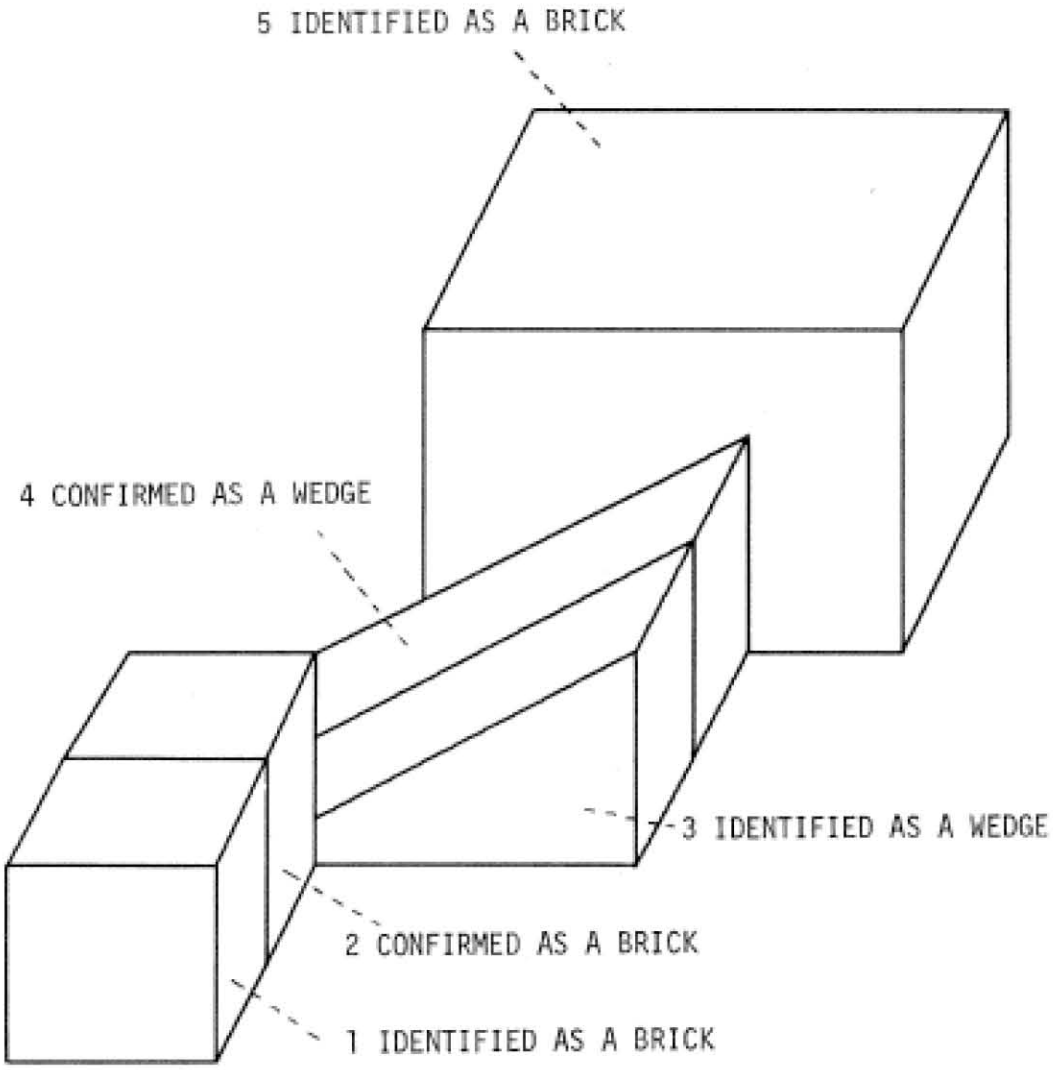

5 IDENTIFIED AS A BRICK
4 CONFIRMED AS A WEDGE
3 IDENTIFIED AS A WEDGE
2 CONFIRMED AS A BRICK
1 IDENTIFIED AS A BRICK

FIGURE 7-25

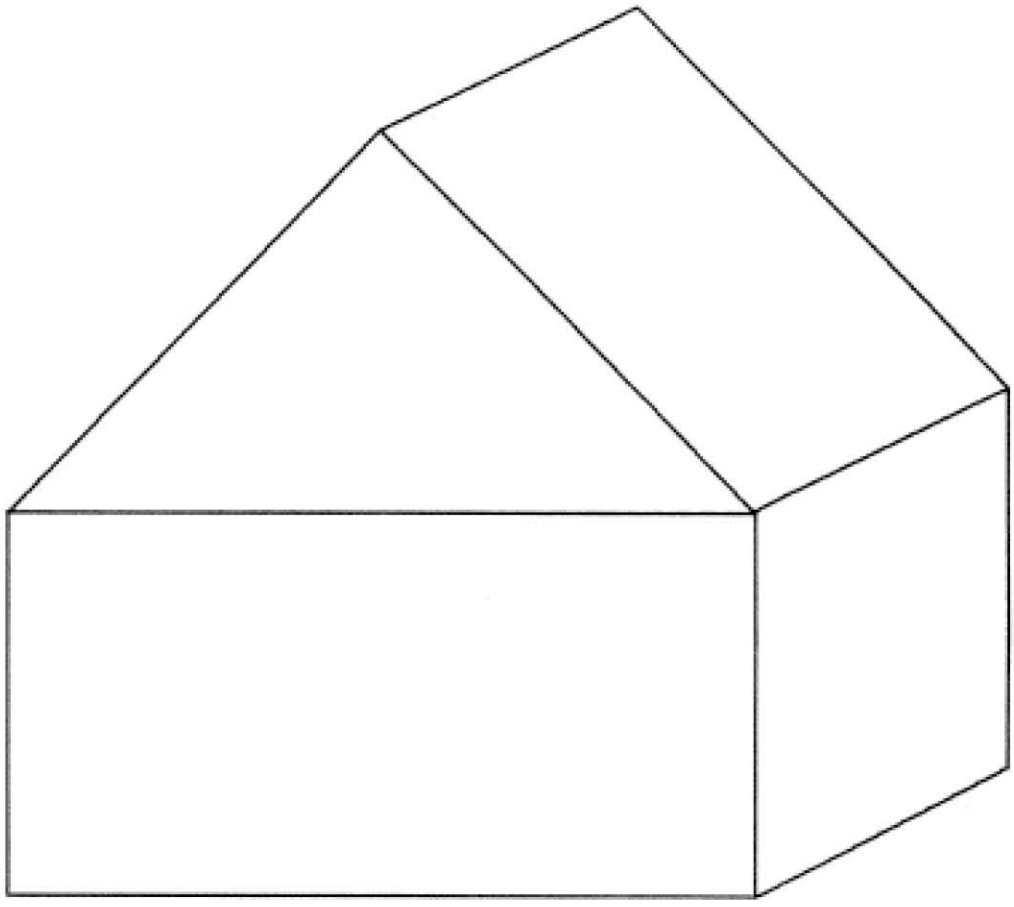


FIGURE 7-26

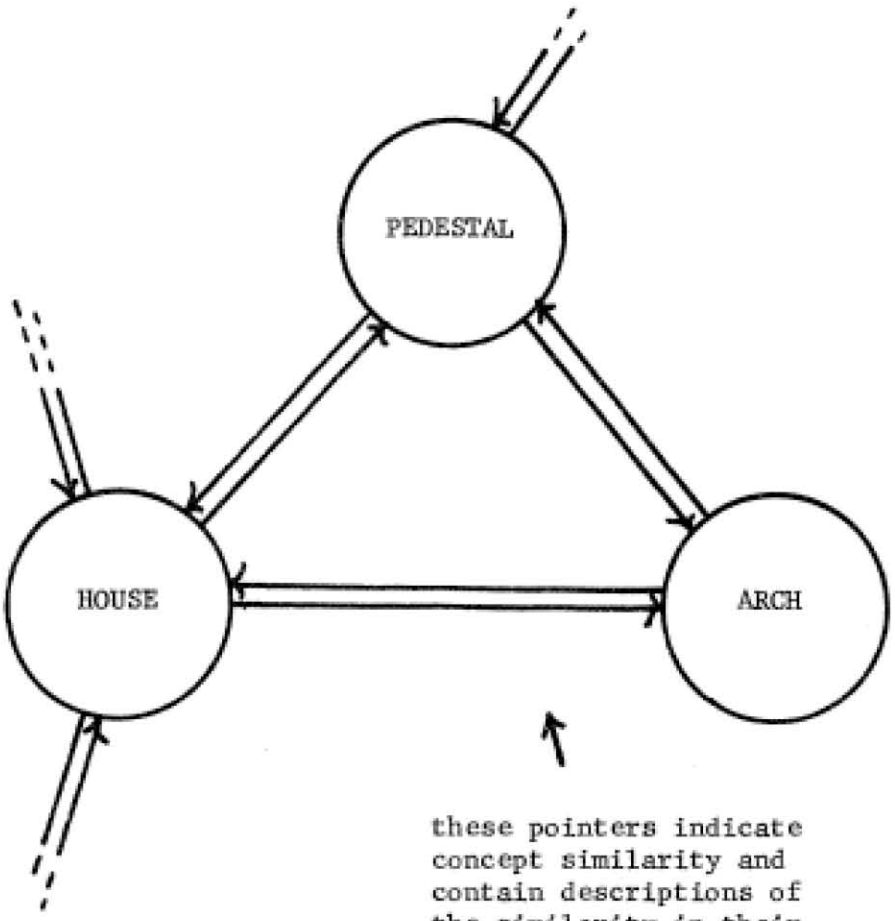

PEDESTAL
HOUSE
ARCH
these pointers indicate concept similarity and contain descriptions of the similarity in their own descriptions.

FIGURE 7-27

the differences between an unknown and a particular model is compared with the descriptions of the similarity net. If the difference between the unknown and a particular model matches the difference between that model and some other model, then identification with that other model is likely.

For example, the scene of figure 7-26 relates to the model of a pedestal in roughly the same way that the model of a house relates to the model of a pedestal. House is consequently elevated to the top of the list of trial models. Figure 7-28 clarifies the procedure.

7.6.1 Similarity Descriptions

The similarity descriptions are simplifications of the comparison descriptions and are part of the description of each pointer that relates similar concepts. When a losing identification reminds the program of some difference structure it has seen, no serious commitment is made and mistaken conjectures do not hurt much. Consequently it is desirable to strip the difference descriptions to the important elements, thereby saving storage space and increasing matching speed, even though some wrong models may be proposed as likely identifications. The simplified description therefore consists in part of a sort of skeleton. (figure 7-29)

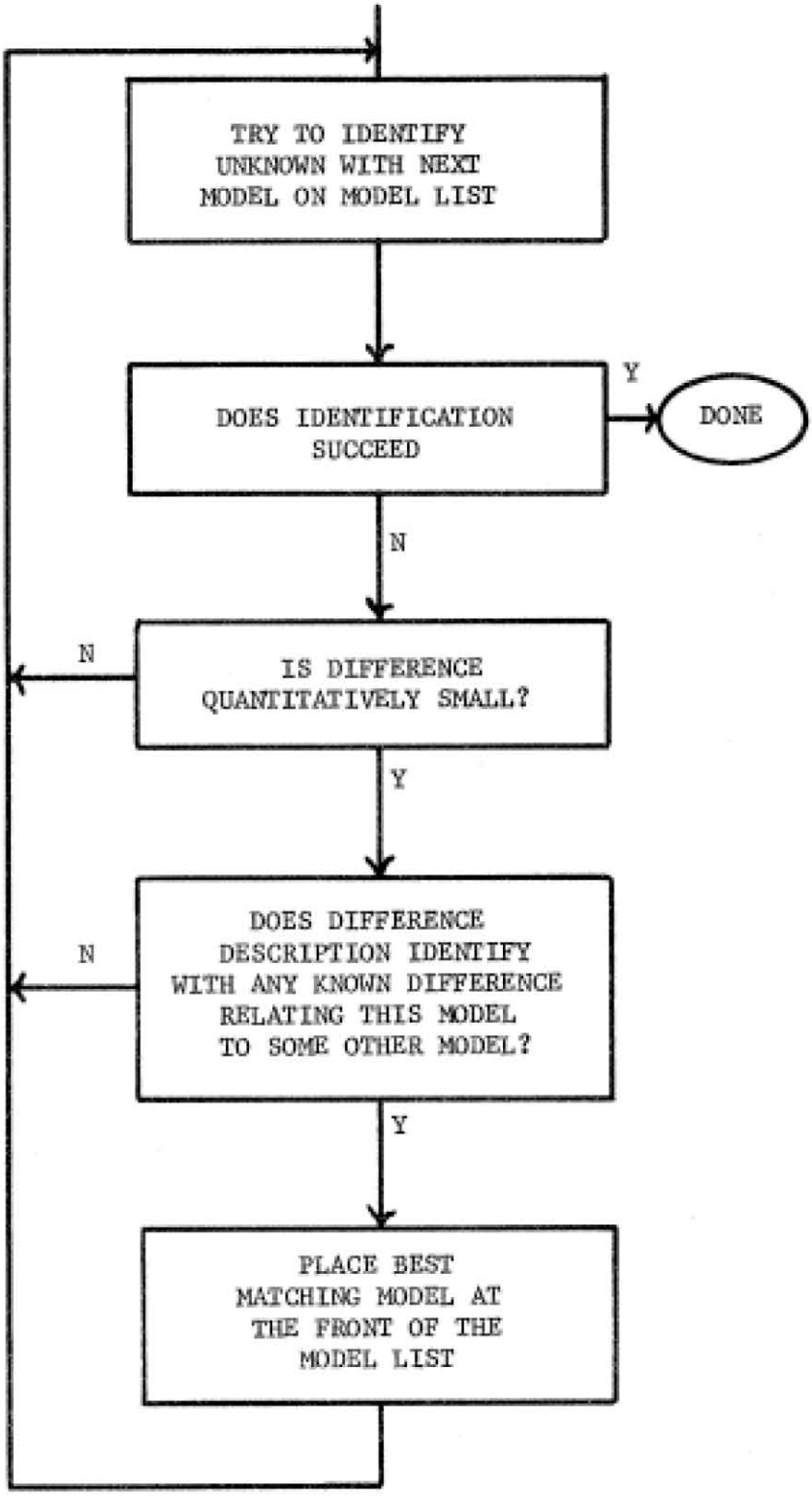

TRY TO IDENTIFY UNKNOWN WITH NEXT MODEL ON MODEL LIST
DOES IDENTIFICATION SUCCEED
Y
DONE
N
IS DIFFERENCE QUANTITATIVELY SMALL?
N
Y
DOES DIFFERENCE DESCRIPTION IDENTIFY WITH ANY KNOWN DIFFERENCE RELATING THIS MODEL TO SOME OTHER MODEL?
N
Y
PLACE BEST MATCHING MODEL AT THE FRONT OF THE MODEL LIST

FIGURE 7-28

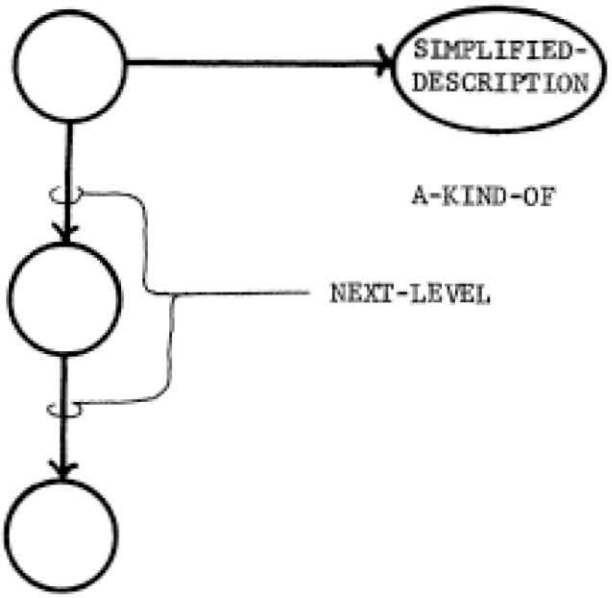


FIGURE 7-29

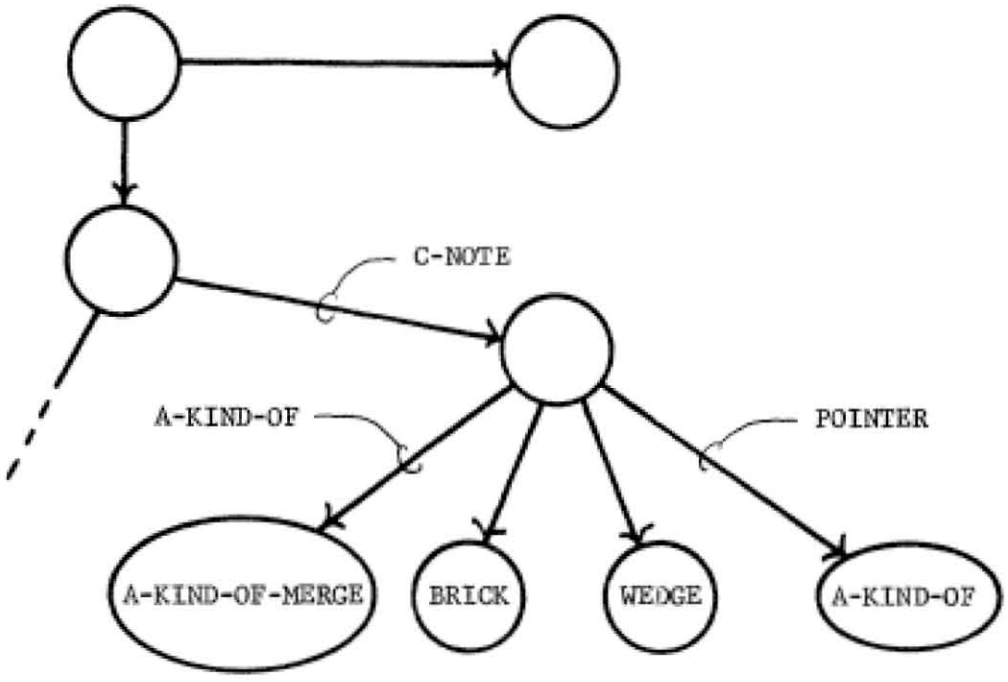


FIGURE 7-30

Intersection c-notes and others associated with positive numbers on the evaluation scale are ignored because only the disruptive c-notes are of interest here. These disruptive c-notes, which suggest poor match, are hung on the skeleton.

Figure 7-30 shows the simplified difference description resulting from comparison of the house model with the pedestal model. Notice that it is exactly the same under this simplification transformation as that resulting from comparison of the pedestal model and the scene in figure 7-26.

7.6.2 Definition of Quantitatively Small

This whole similarity scheme depends on the fact that two models may have only one or a few differences that make them strongly different in a qualitative sense. Indeed, the similarity links should exist only when the models involved are reasonably close in the sense of producing few differences. When this is true, an unknown that nearly identifies with one model in the sense of few differences is assured of matching well with the other, particularly when the two sets of differences match. Thus there must be some rule for deciding if the number of differences is sufficiently small to warrant a pair of pointers in the similarity network. Currently the machine

considers sufficiently small to mean the number of mismatch-causing c-notes is either less than two or less than one-third of the number of other c-notes.

7.7 The Needle in the Haystack

The scene of figure 7-31 is curious in that one can find an arch, a pedestal, a house, and a tent in it if one is looking for them. But if they are not specifically searched for, mention of these particular models is unlikely to appear in a description of the scene. Although the configurations are present, they are hidden by extraneous objects so well that general grouping programs are unlikely to sort them out. Yet the question, "Does a certain model appear in the scene?" is certainly a reasonable one. One way to attack it divides nicely into three parts:

1. Find those objects in the scene that have the best chance of being identified with the model. If the model has unusual pointers or references unusual concepts, the program pays particular attention to them. Similarly, extra attention is paid to the emphasized parts of the model, for if mates cannot be established for them, solid identification cannot be affirmed. Happily, my standard network matching program does these things without augmentation. The result is a set of links between the objects of the model and their nearest analogues in the scene. The other parts of the scene remain unlinked and end up appearing in exit c-notes.

2. Once a good group of objects is picked, then the pointers relating these objects to the other

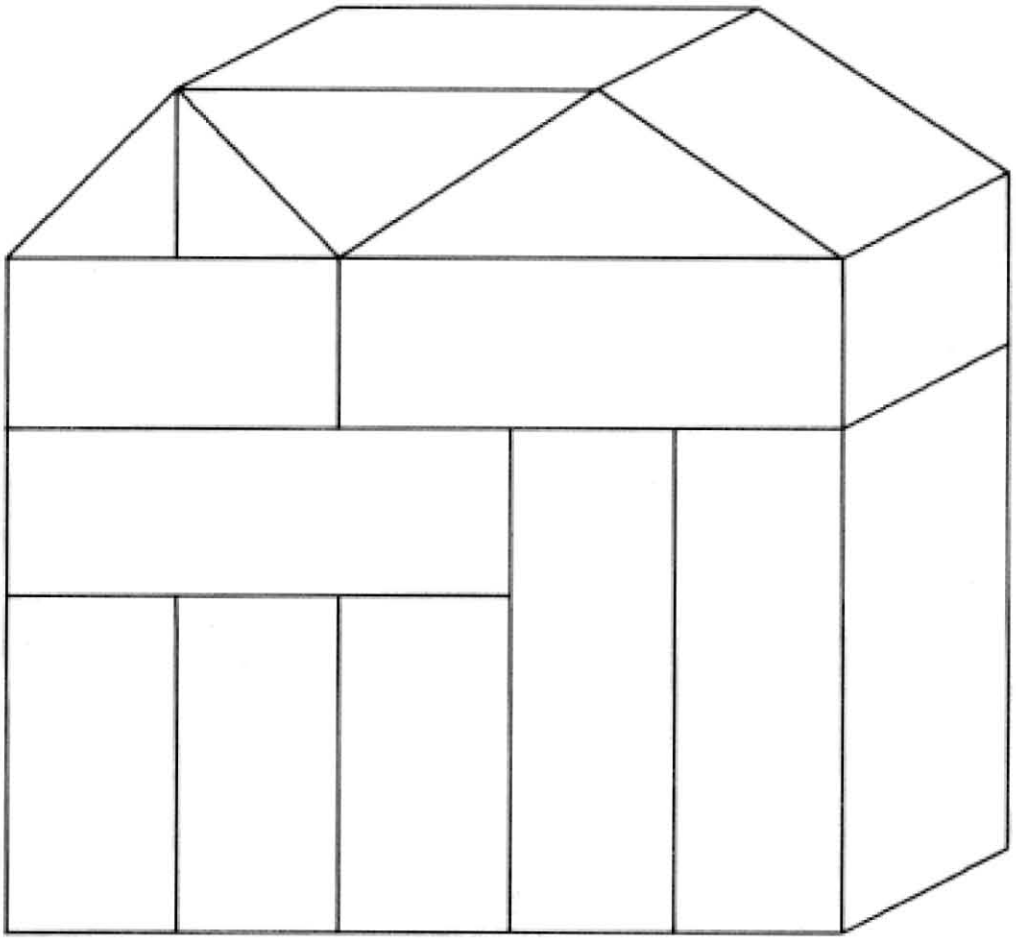


FIGURE 7-31

objects in the scene are temporarily forgotten. In human terms, this is like painting the subgroup a special color or otherwise reducing possible confusion from relations with the surrounding objects.

3. Finally, with the best group of objects set into relief by the previous excision, the ordinary identificaton routines are applied with the expectation of reasonable performance.

The folly of direct application of the identification programs lies in the myriad irrelevant exit c-notes that the extra objects in the scene would cause. Such clutter leaves the machine as bewildered as it does humans.

7.8 Reacting to Identification

Once identification of a substructure happens, the discovery should contribute to the store of knowledge. Figure 7-32 illustrates some of the more obvious things done after identification of the house and arch in figure 7-33. The highest level node no longer connects directly to the individual objects. Instead, those objects dangle from new subscene nodes by ONE-PART-IS pointers. Similarly the old top level concept points to the new subscenes with ONE-PART-IS. The subscene nodes naturally point by A-KIND-OF to the models they identify with.

Rather more can be done if the machine knows something about how the relations of a group's components

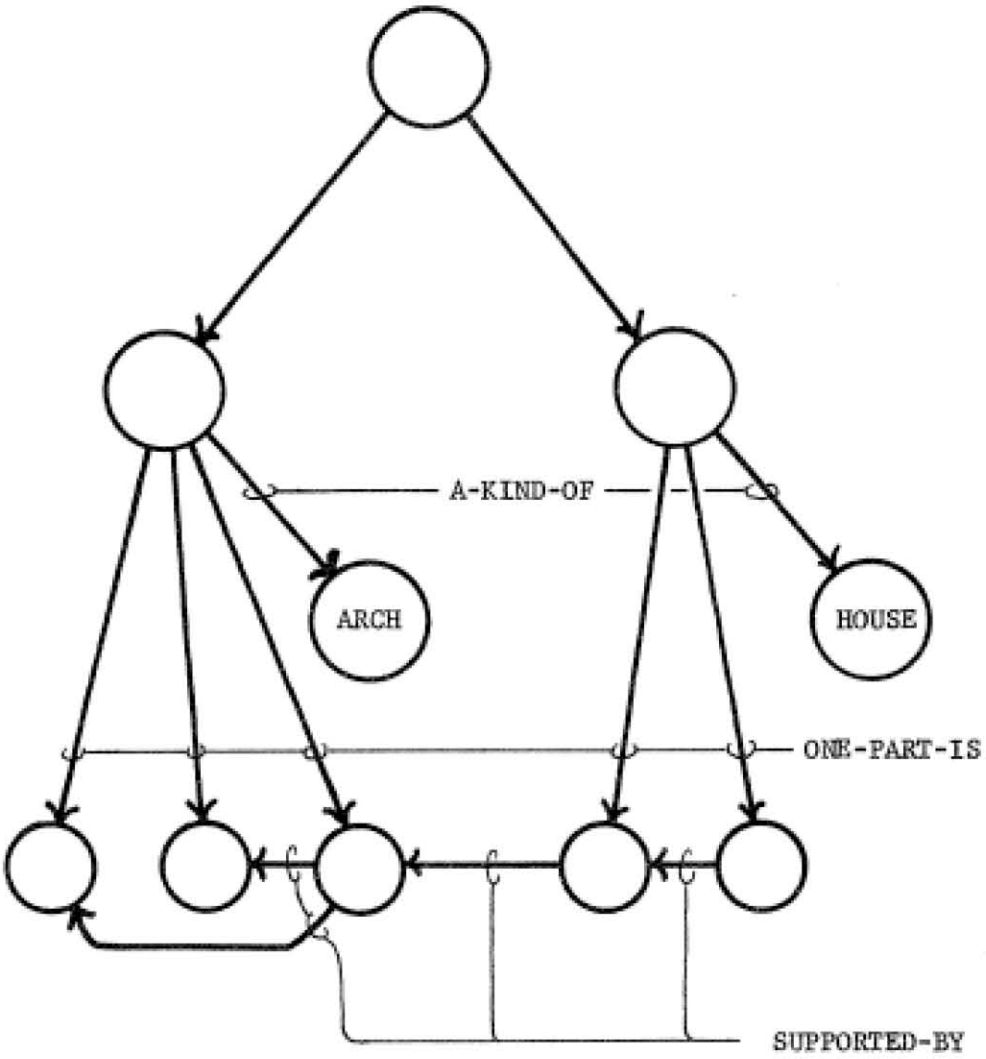


FIGURE 7-32

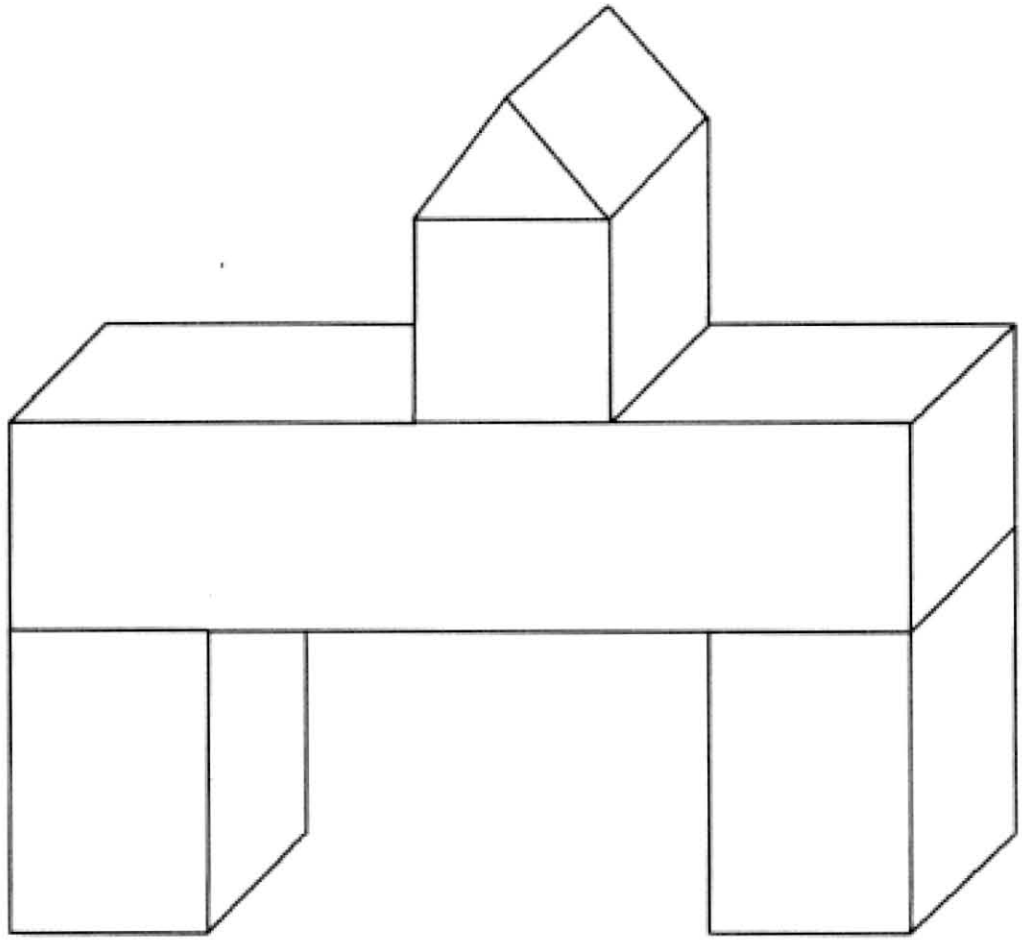


FIGURE 7-33

to external objects dictate the relations of the group. Any knowledgeable machine knows that a house configuration rests on whatever its bottom object rests on. More generally, the following rule seems reasonable:

> Suppose A and B are groups of objects identified as substructures, then if an object of A relates to an object of B by SUPPORTED-BY, then substructure A relates to substructure B by SUPPORTED-BY.

In consequence, the net in figure 7-32 becomes that in figure 7-34.

7.8.1 Examples

Using this same procedure, the scene in figure 7-35 soon reaches the state of illumination shown in figure 7-37. By examining either the picture or the net, it is easy to see that the arches AT, AL, and AR themselves constitute a sort of super-arch with arches as parts instead of objects. The machine does not refuse this substitution since the model for ARCH has only ONE-PART-IS pointers to BRICK and OBJECT, not ONE-PART-MUST-BE pointers. The matching score is simply lower than it would be for arches made of bricks. The final description essentially states that the scene consists of a sort of arch supported by an arch composed of three arches.

Figure 7-36 shows a richer example including instances of a pedestal. Again the machine identifies

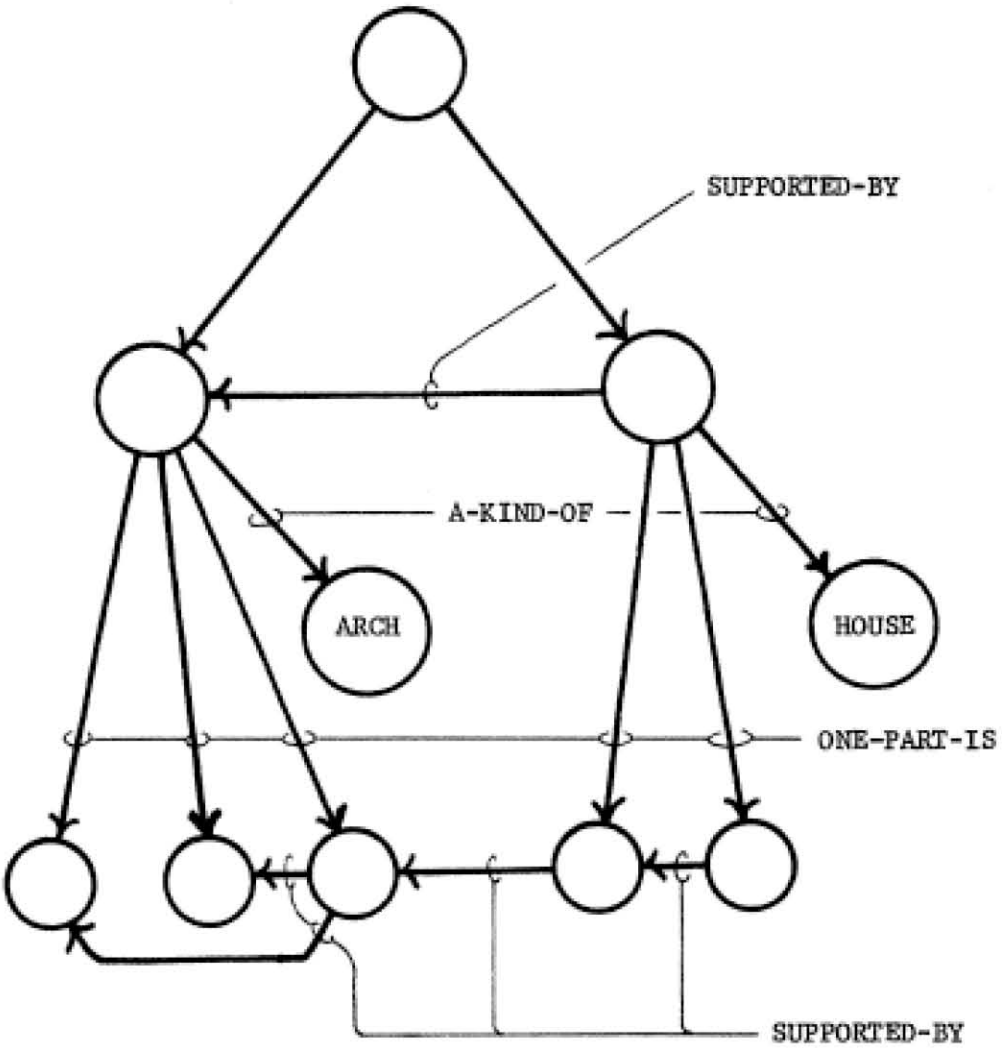


FIGURE 7-34

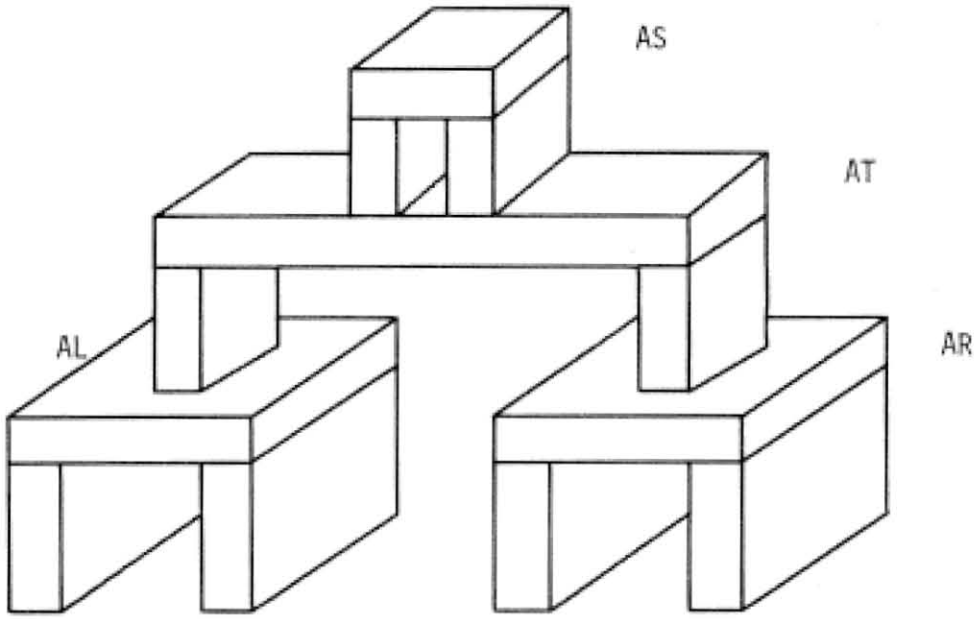


FIGURE 7-35

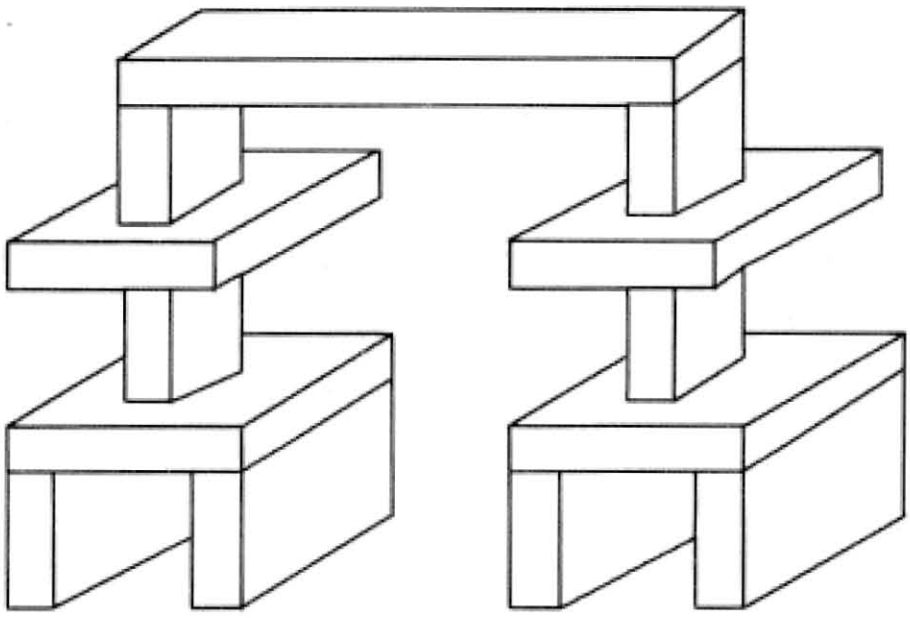


FIGURE 7-36

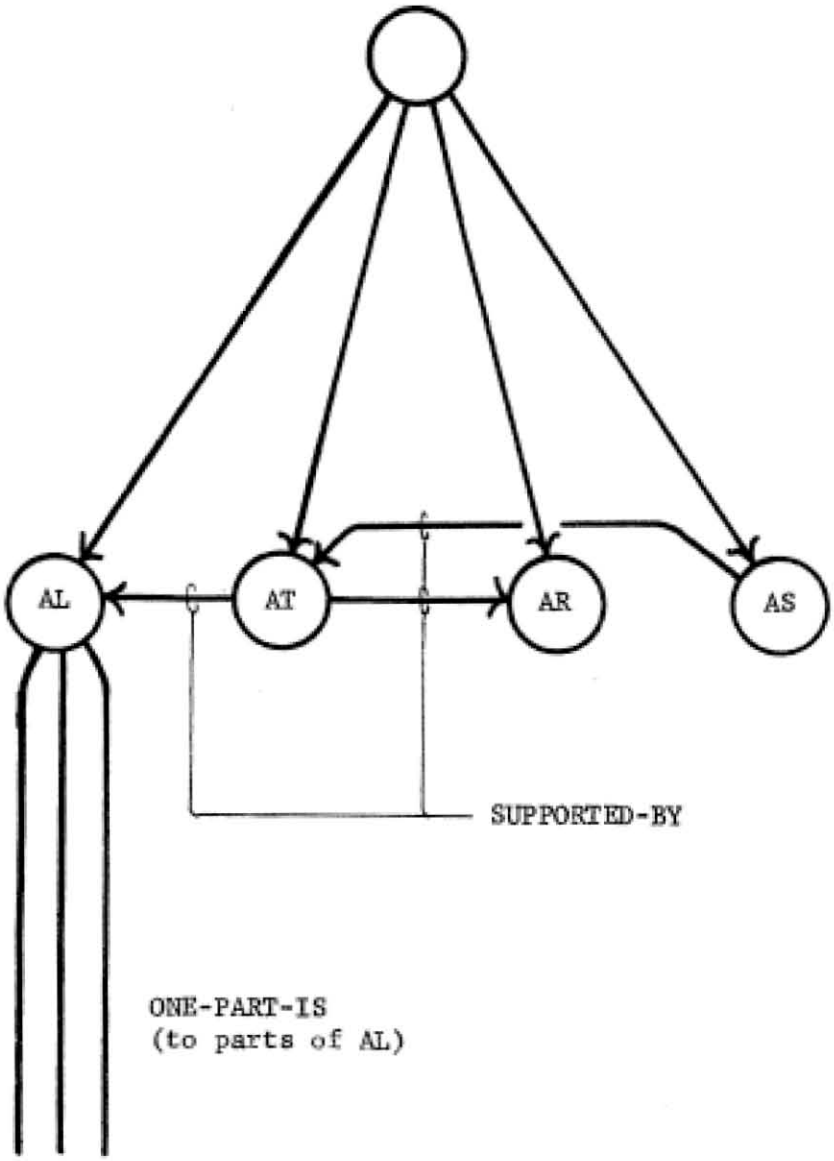


FIGURE 7-37

groups, establishes relations between the identified groups, and then tries to identify groups of groups, reporting eventually that there is an arch composed of an arch on top with two pedestals for supports. It then notices that this generalized arch is supported by two ordinary arches. But the generalized arch on top of two supporting arches again is a kind of arch, the fifth and last discovered.

7.8.2 Other Relations

I have not thought much about the calculation of other group properties. It seems reasonable, however, that a set of programs using the following ideas should work to some extent, albeit crudely. To find IN-FRONT-OF relations between groups one can use the above rule for SUPPORT with the obvious exchange of IN-FRONT-OF for SUPPORTED-BY. To establish the size of a group one can add together the individual areas of its objects. To check for LEFT-OF and RIGHT-OF, one uses the center of area and extreme points of the entire group rather than those of an individual object. But otherwise the left-right algorithm may remain the same.

8 Closing Remarks

8.1 A System

The flow diagram in figure 8-1 shows how the techniques fit together with those of others to form a primitive scene-perceiving system. At the very beginning lies the scene, from which all information ultimately derives. A program developed by Griffith [7] watches the scene through an eye resembling television camera. The result is a line drawing. Next programs of Mahabala [1] and Guzman [2] classify vertexes and group regions into bodies. Next is a stage in which object identification is done. Following closely, one has the determination of object-object relations and then group identification. Finally there is identification of group-group relations.

Beyond this, action depends on intent. On one path one finds attempted identification of the entire scene with a known model or models. On another, an effort is made to find an instance of some particular model in the scene. Still another path involves use of the description to help form new concepts.

8.2 Conclusions

This collection of ideas and techniques supports four major contentions, each of which depends on those preceding it. These things are small steps for a man, but

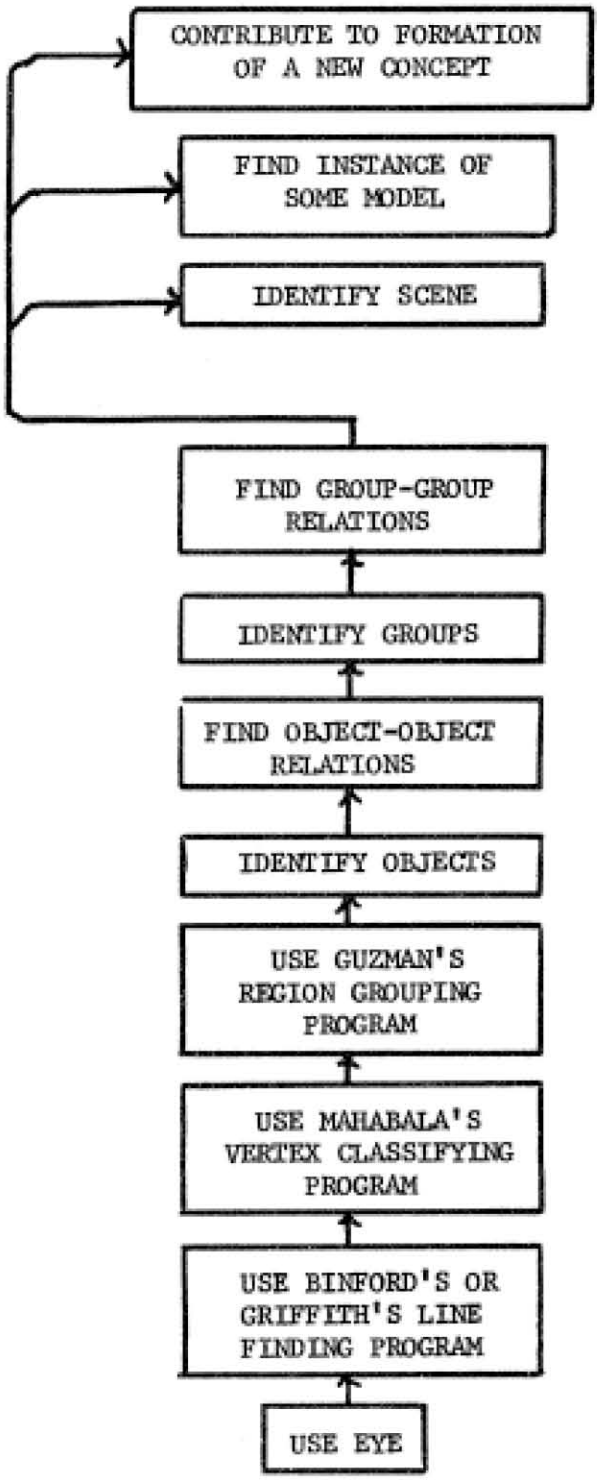

CONTRIBUTE TO FORMATION OF A NEW CONCEPT
FIND INSTANCE OF SOME MODEL
IDENTIFY SCENE
FIND GROUP-GROUP RELATIONS
IDENTIFY GROUPS
FIND OBJECT-OBJECT RELATIONS
IDENTIFY OBJECTS
USE GUZMAN'S REGION GROUPING PROGRAM
USE MAHABALA'S VERTEX CLASSIFYING PROGRAM
USE BINFORD'S OR GRIFFITH'S LINE FINDING PROGRAM
USE EYE

FIGURE 8-1

giant leaps for a machine.

1. A computer can produce a detailed scene description consisting of the same sort of facts humans observe.

2. These descriptions lead in turn to descriptions of how scenes compare with one another.

3. An understanding of how scenes compare permits the computer to learn models for new concepts from examples and leads to a new way of thinking about learning.

4. These models finally endow the machine with the ability to recognize instances of previously learned concepts.

8.3 Background Issues

In a more cosmic sense, the goal behind this work is to make a machine that can understand the environment just as we humans seem to. Some critics of Artificial Intelligence think that this is not possible, perhaps because they cannot imagine how it can be done. I think the real hang up must lie in the understanding one has about the notion of understanding. A review of a few dictionaries convinces me that the editors are hard pressed to define the word without using it. It is as if it were a word so basic that it cannot be described in simpler terms.

But surely to understand must involve the formation of a descriptive plateau of knowledge lying somewhere between raw, totally unprocessed data and detailed answers

to problems. I do not wish to belabor this point, but I feel that the sort of abstraction represented by the network description of a scene can be viewed as constituting a sort of understanding. If so, depth of understanding corresponds roughly to the elaborateness of a description, dense networks suggesting more understanding than sparse ones.

Another notion of long standing concern to philosophy is that of the ideal form. Yet little work seems to have gone into careful study of what humans mean by such simple concepts as that of the TABLE. I believe study and improvement of the concept generator constitutes a fresh approach to this problem and may lead to interesting new results.

8.4 Suggestions for Further Work

Improvements to and extensions of this work can be understood in terms of two extremes: minor change and major overhaul. The minor-change category is large because the highest priority goal in such work must be a complete system. A complete system, however flimsy, serves to guide resource allocation into the most deserved problem areas. Without experience with such a system, one risks suffering from the phenomenon of diminishing returns, expending great effort for marginal improvements

on relatively strong pieces of the system. But the natural result is that there is much room for further improvement of the system's parts. Some possibilities have already been mentioned, but it is appropriate to mention them here as a convenience for those who wish to work in the area.

1. Nearly all the programs that establish relations between objects can be improved. The program that looks for support blunders sometimes because obvious bottom lines are overlooked and sometimes because a scene has tipped background objects. The program that looks for in-front-of relationships cannot handle situations in which objects are aligned. Many of these programs could benefit from a program that could imagine hidden lines.

2. No distinctions are made between obvious, unarguable properties and borderline cases. It might be good if the analytic programs could report things like certainly-left-of or sort-of-left-of instead of invariable, undifferentiated left-of.

3. The rules for the identification of a scene with a model need refinement. The weighting associated with the various differences and the way those weights are combined have a specious quality. It would be fine if some way could be devised to eliminate the numbers altogether, perhaps through a more intelligent program with a built-in understanding of priorities.

4. The schemes for recognizing reasonable clusters of objects is particularly primitive and has undergone too little testing. Mechanisms must be found for producing and handling alternatives to the first partition devised.

5. The entire concept generation procedure and its ramifications certainly should absorb great

attention. More powerful methods for recognizing important differences are needed. The machine should have a faculty by which it can complain or ask questions if the teacher is doing poorly. Generalization to functional properties is a must.

6. The network matching program, although it is at the core of the entire system, is a hastily programmed, slow and stubborn stumblebum. An improved version would simultaneously increase the power of the many system activities that use it.

The other kind of improvement is the major overhaul. This is not aimed at sophisticating an existing part of the system, but rather focuses on the spectacular increases in power derived from bolder, bigger ideas. I discuss two such possibilities below. Regretably, these problems are difficult to formulate and delimit.

First is the general problem of introducing bi-directional information flow. Glance again at the flow diagram of figure 8-1. Notice that all of the arrows point from bottom to top, indicating that all information moves in one direction only. There is as yet no way a process can discourse with and modify the behavior of any process acting below it.

It seems clear that for high level command to affect low level operation in a non-trivial way, there must be some alternative or additional method that can be thrown into battle. As yet few such points of influence exist in my system. There are two left-of -- right-of procedures,

and there is the option of using or not using various relation-finding and grouping procedures. But the forte of an internally interactive system will probably involve selection of one method from several possibilities among which there are trades between speed and accuracy.

Another amorphous problem is that of wedding the visual capabilities of this system with other systems that specialize in different kinds of perception. A real robot should understand the environment not only in terms of vision but also in terms of touch, sound, language, and perhaps other mediums. Understanding each of these is a major problem, but as work proceeds, there will be the super-problem of understanding how various perceptions of the environment should interact to form a unified understanding.

> I understand neuroanatomical evidence is that evolution has come to grips with this problem only lately and only in man with any finesse. Norman Geschwind reports that monkeys have very limited ability to correlate things they learn via the visual, auditory, and somesthetic senses [8]. Indeed it may be reasonable to say each monkey is really three monkeys occupying the same skull, a visually oriented monkey, a auditory one, and a somesthetic one. Man apparently avoids this through a chunk of cortex that somehow matches up these perceptions. The chunk believed responsible by Geschwind is called the inferior parietal lobule.

Appendix

This appendix is a cursory introduction to the network matching program essential to many of the system's operations. Its job is to determine which nodes of two descriptions best correspond. The corresponding nodes are said to be the linked pairs.

The program starts with the entry nodes of two descriptions. It immediately inquires if there is evidence that the two nodes should be considered a linked pair. The answer is yes if both nodes have a common pointer to some common intersection node. Thus X and X' are a linked pair in figure A-1, but they are certainly not in figure A-2, since nodes X and X' have neither pointers nor nodes in common.

If no linking can occur, the program moves down one level through the most common pointer present to daughters of the currently inspected nodes and tries to find linked pairs among them. In figure A-3, the entry nodes are not linked on first inspection since there are no common pointers to any common node. They both have daughter nodes, however, and these are next examined. P is the most numerous pointer, so nodes C1, C2, C1', and C2' form two groups in which the program tries to find good pairs

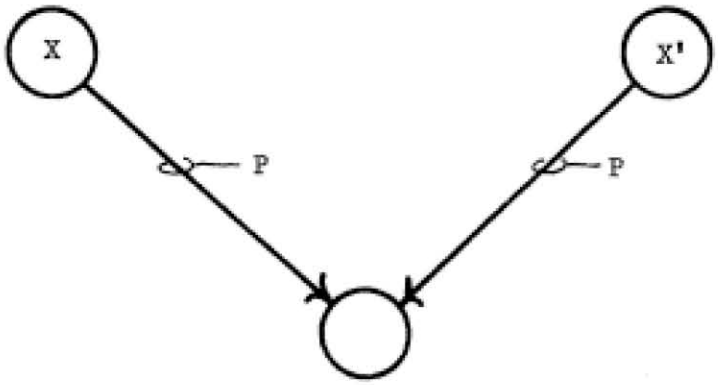


FIGURE A-1

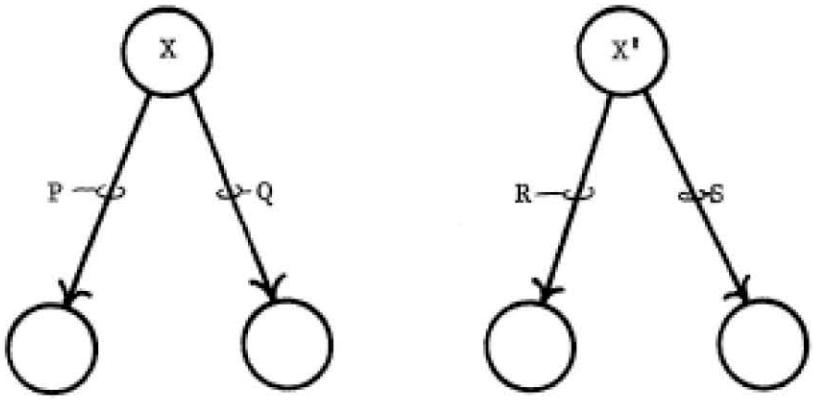


FIGURE A-2

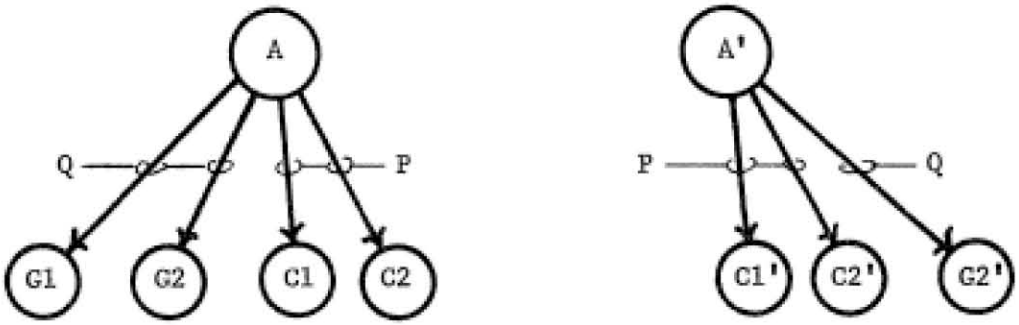


FIGURE A-3

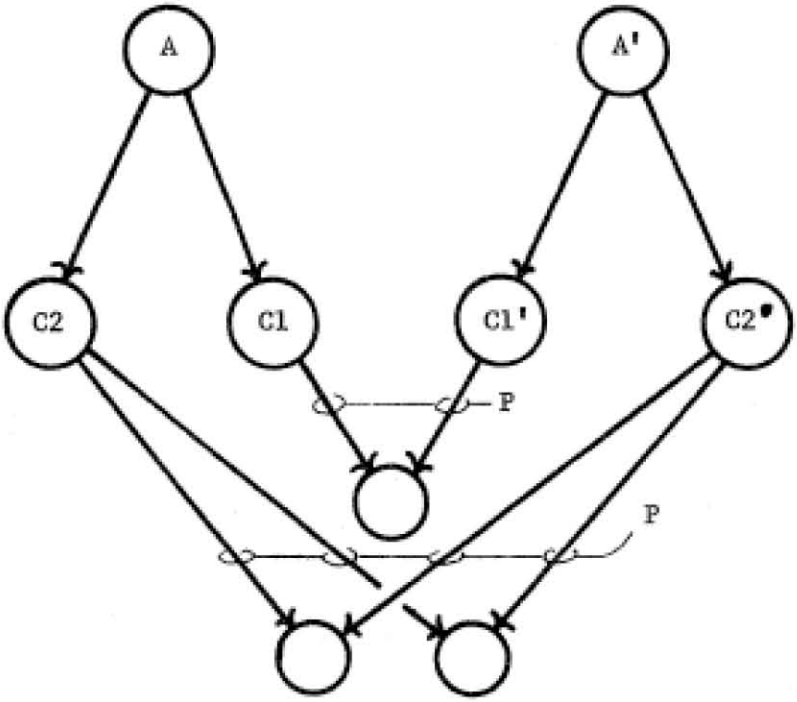


FIGURE A-4

to establish as linked. (figure A-4) Since C1 and C1' both have the same pointer to a common node they are good candidates for the formation of a linked pair. Nodes C2 and C2' are even more alike, however, since they have two pointers to two common nodes. Consequently, C2 and C2' are linked first, with action on C1 and C1' postponed.

Each time a pair of nodes is linked, the program trys to link up any of their daughter nodes that are not intersections, In the example this means that an effort is made to find a linked pair between the left node, E1, and the right nodes, E1' and E2', all of which are found at the end of P pointers. (figure A-5) This is the first example of a contest. Both E1' and E2' share common intersections with E1. The winning pair picked by the machine is always the one with the highest count of common pointers to intersectons or previously linked pairs. In this case, E1-E1' scores higher because there is not only a set of pointers to the intersection node, but also a set to the previously linked pair C2-C2'.

The linking of E1 and E1' causes examination of their daughters, F1 on the left, F1' and F2' on the right. (figure A-6) F1 and F2' both have a pointer R to a common node. F1', however, has the must-be version of the pointer R to the same node. Such satellites occur

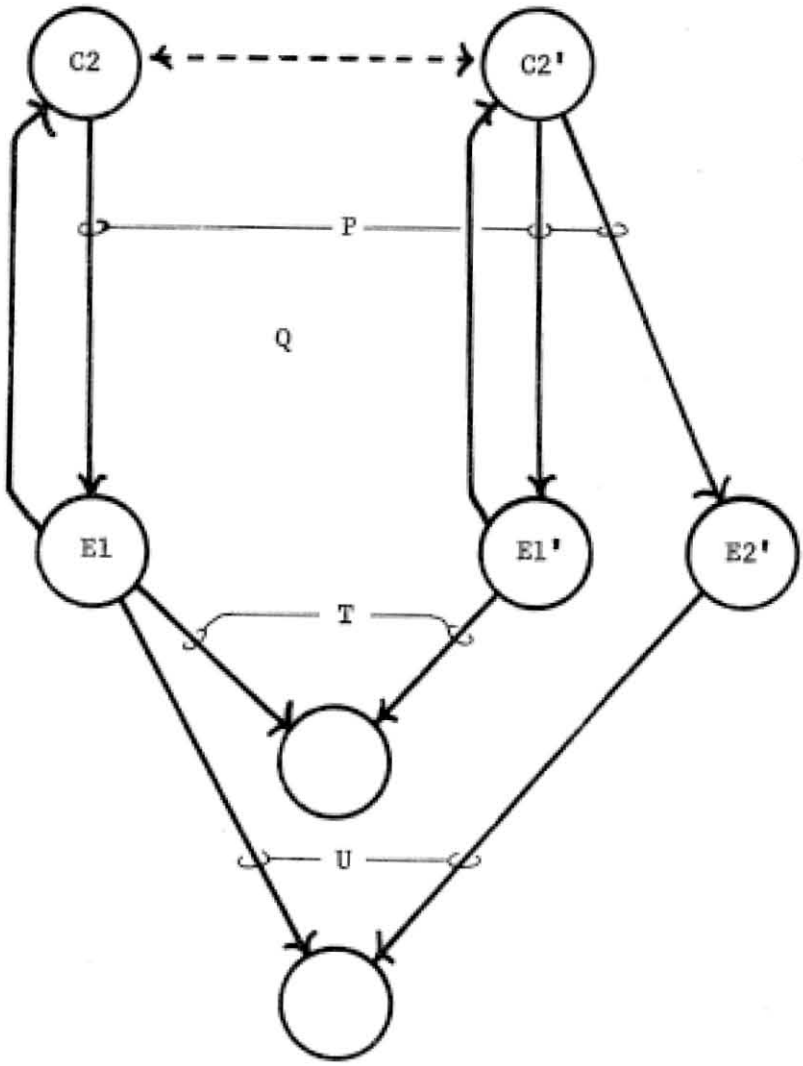


FIGURE A-5

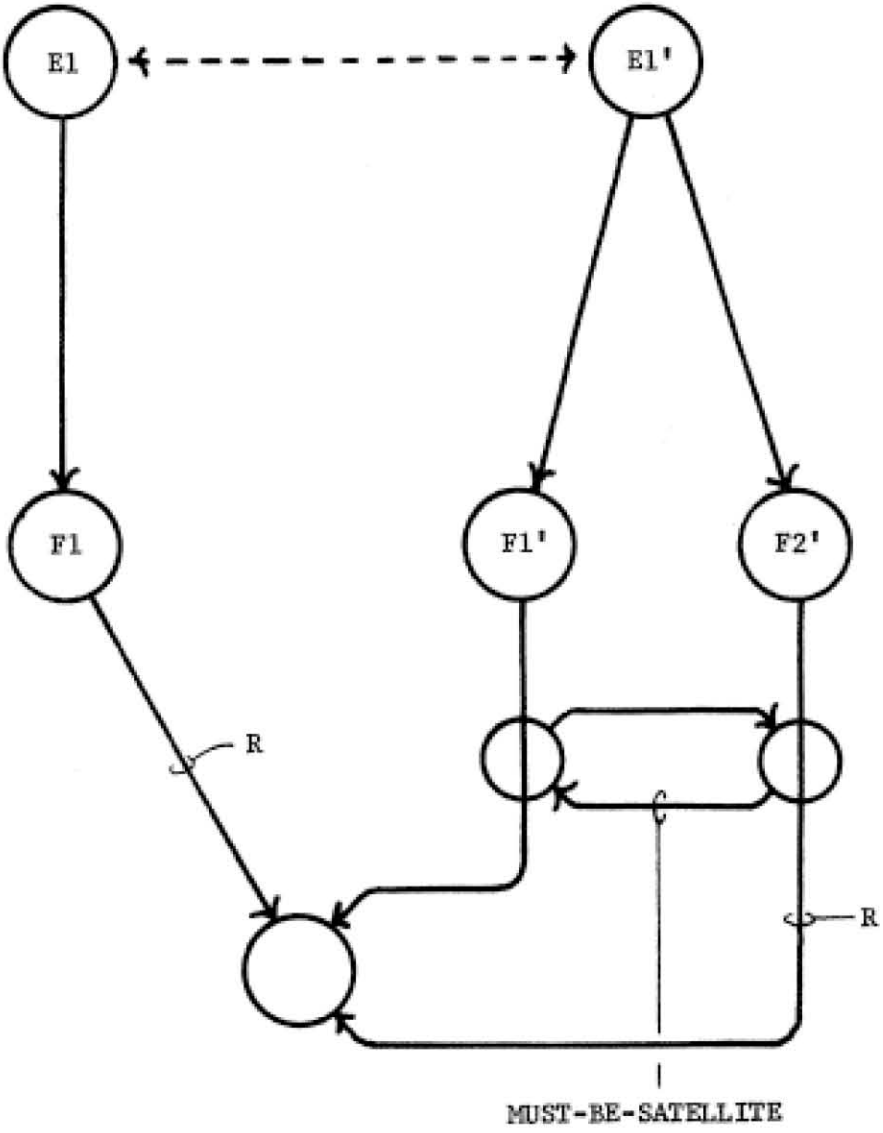


FIGURE A-6

frequently in models, indicating mandatory relations. Priority is given to matches in which satellites correspond to the pointers they are modifications of. This means the R with MUST-BE-R pointer match between nodes F1 and F1' has somewhat more weight than the match of R with R that one would have between F1 and F2'. The F1-F1' match therefore is considered the better one and results in a linked pair.

F1 and F1' have no daughters to be paired so no further penetration of the net occurs. The next job is to re-examine the next higher parent nodes because links just formed may provide enough new evidence to link two higher nodes. In this case backup first considers nodes E1 and E1'. E1 and E1' are already linked. Attention therefore pops up still another level to C1 and C1'. C1 and C1' are linked. Re-examination of their unlinked daughters, D and E2, reveals nothing new.

Once more the programs attention shifts upward, this time to A and A'. Now there is a pair of pointers to linked nodes supplying linking evidence. (figure A-7) A is consequently linked to A'.

Next comes further examination of the remaining daughters of A and A'. Q is now the most common pointer to unaccounted for nodes, pointing as it does to G1, G2,

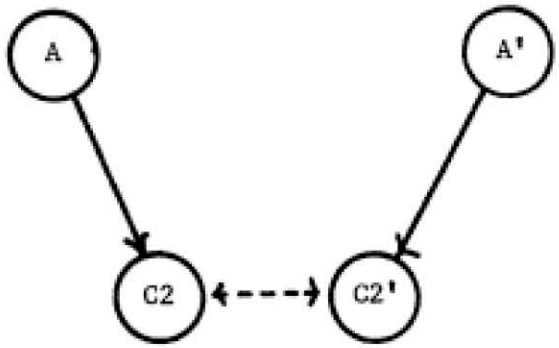


FIGURE A-7

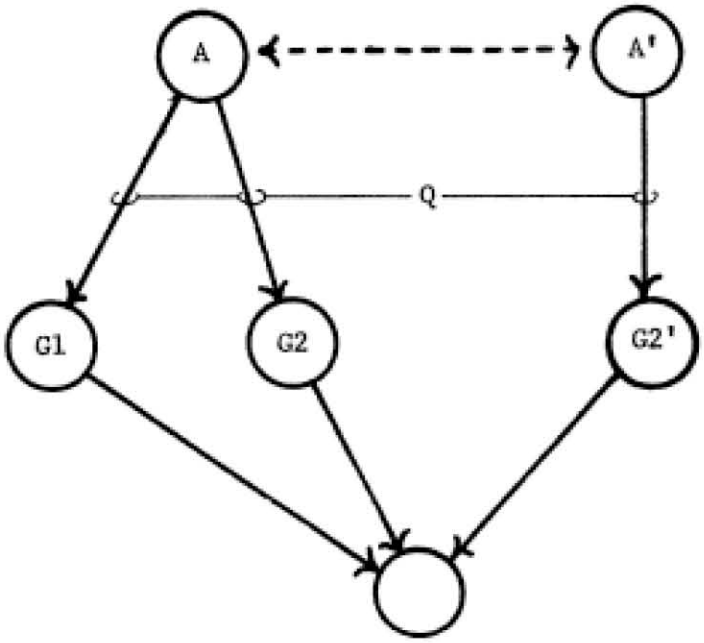


FIGURE A-8

and G2'. (figure A-8) Among these nodes there is the same amount of evidence for linking G1 and G2' as there is for linking G2 and G2'. When this is the case and no further evidence can be collected from linking lower level nodes, a linking is selected randomly from those possible. Here G2 and G2' are linked.

This leaves only reexamination of C2 and C2'. The intersection evidence is clear and they are linked. Since A and A' have no more daughters and since they are the entry nodes, the process terminates reporting the linkages indicated on the fully displayed network of figure A-9.

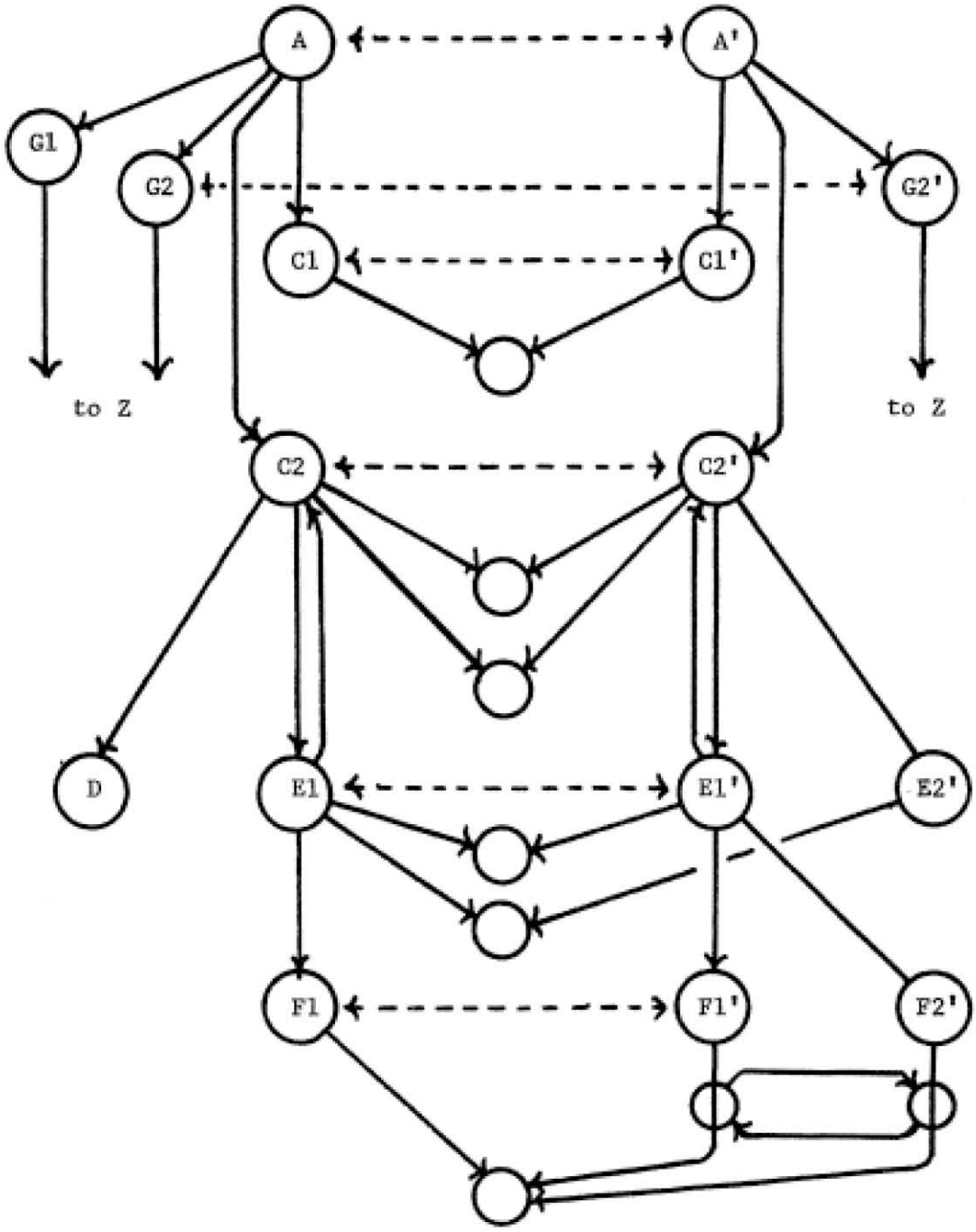


FIGURE A-9

References

[1] H. N. Mahabala, "Preprocessor for Programs which Recognize Scenes," Artificial Intelligence, Memo. No. 177 (Cambridge, Mass.: Project MAC, MIT, August 1969).

[2] Adolfo Guzman, "Computer Recognition of Three-Dimensional Objects in a Visual Scene," Report MAC-TR-59 (Thesis)(Cambridge, Mass.: Project MAC, MIT, December 1968).

[3] Jean Piaget, The Child's Conception of Number (New York: Humanities Press, 1952).

[4] Thomas G. Evans, "A Heuristic Program to Solve Geometric-Analogy Problems," Ph.D. Thesis, Department of Mathematics, MIT (Cambridge, Mass.: MIT, May 10, 1963).

[5] George W. Ernst and Allen Newell, GPS: A Case Study in Generality and Problem Solving (New York: Academic Press, 1969).

[6] Allen Newell, "Learning, Generality, and Problem-solving," Proceedings of the IFIP Congress: 407-412 (Amsterdam: North Holland Publishing Co., 1969).

[7] Arnold Griffith, "Computer Recognition of Prismatic Solids, Ph.D. Thesis, Department of Mathematics, MIT (Cambridge, Mass.: MIT, June 1970).

[8] Norman Geschwind, Disconnexion Syndromes in Animals and Man. Brain 88: 237-294, 585-644. 1965.

Bibiliography

Ernst, George W. and Newell, Allen. GPS: A Case Study in Generality and Problem Solving. New York: Academic Press, 1969.

Evans, Thomas G. "A Heuristic Program to Solve Geometric-Analogy Problems." Ph.D. Thesis, Department of Mathematics, MIT. Cambridge, Mass.: MIT, May 10, 1963.

Geschwind, Norman. Disconnexion Syndromes in Animals and Man, Part I. Brain 88: 237-294. June 1965.

Geschwind, Norman. Disconnexion Syndromes in Animals and Man, Part II. Brain 88: 585-644. Sept. 1965.

Griffith, Arnold. Computer Recognition of Prismatic Solids. Ph.D. Thesis, Department of Mathematics, MIT. Cambridge, Mass.: MIT, June 1970.

Guzman, Adolfo. "Computer Recognition of Three-Dimensional Objects in a Visual Scene." Report MAC-TR-59 (Thesis). Cambridge, Mass.: Project MAC, MIT, December 1968.

Mahabala, H. N. "Preprocessor for Programs which Recognize Scenes." Artificial Intelligence, Memo. No. 177. Cambridge, Mass.: Project MAC, MIT, August 1969.

Newell, Allen. "Learning, Generality, and Problem-solving." Proceedings of the IFIP Congress: 407-412. Amsterdam: North Holland Publishing Co., 1969.

Piaget, Jean. The Child's Conception of Number New York: Humanities Press, 1952.