Winston (ed), The Psychology of Computer Vision

5 LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick Henry Winston

5.1 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 set of working computer programs that sheds some light on these questions by demonstrating how a machine can be taught to see and learn new visual concepts. The programs work in the domain of three-dimensional structures made of bricks, wedges, and other simple objects.

Centrally 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 small number of significant points. These near misses prove to convey the essentials much more directly than repetitive exposure to ordinary examples.

Good descriptions are equally important. I believe learning from examples, learning by imitation, and learning by being told uniformly depend on good descriptions. My system therefore necessarily has good methods for scene description and description comparison.

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.

Fig. 5.1

5.1.1 Scene Description and Comparison

Much of the system to be described focuses on the problem of analyzing toy block scenes. There are two very simple examples of such scenes in Fig. 5.1.

From such visual images, the system builds a very coarse description as in Fig. 5.2. Then analysis proceeds, inserting more detail as shown in Fig. 5.3. And finally there is the very fine detail about the surfaces, lines, vertexes, and their relationships.

Such descriptions permit one to match, compare, and contrast scenes through programs that compare and contrast descriptions. After two scenes are described and corresponding parts related by the matching program, differences in the descriptions can be found, categorized, and themselves described. 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.

The program that does this must be able to examine the descriptions of Fig. 5.3 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. Of course the matcher 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.

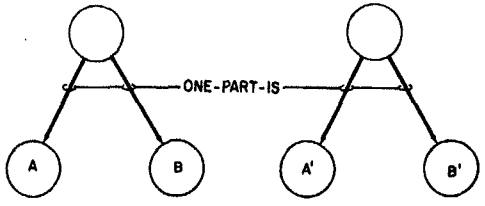


Fig. 5.2

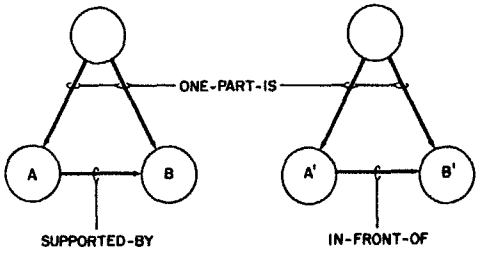


Fig. 5.3

5.1.2 Learning and Identification

To build a machine that can analyze line drawings and build descriptions relevant to some comparison procedure is interesting in itself. But this is just a step toward the more ambitious goal of creating a running program that can learn to recognize structures. I will describe a program that can use samples of simple concepts to generate models.

Figure 5.4 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 Fig. 5.4(*a*). 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 Fig. 5.4(*b*). It learns that it does not matter much what the top object is

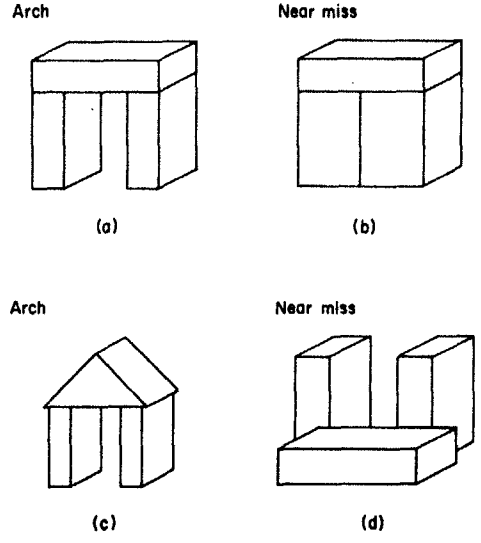


Fig. 5.4

from Fig. 5.4(*c*). And then from Fig. 5.4(*d*) it learns the fact that for one object to be supported by the others is a definite requirement, not just a coincidence applying to 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.

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 ways. 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 there is a 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 either sensitive to context or prejudiced toward locating a particular type of object. Elementary algorithms for both of these kinds of identifications are discussed later.

5.1.3 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 should 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 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 seems characteristic of human thought.

Another reason for designing programs that describe scenes in human terms is that human judgment 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.

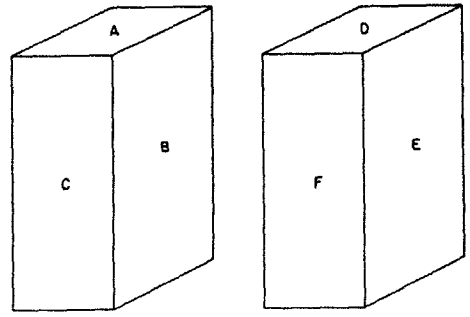


Fig. 5.5

5.2 BUILDING DESCRIPTIONS

The network seems to have the appropriate blend of flexibility and simplicity needed to deal straightforwardly 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. In Fig. 5.5, for example, one has concepts such as OBJECT-ABC and OBJECT-DEF. These are represented diagrammatically as circles in Fig. 5.6. Labelled arrows or pointers define the relationships between the concepts. Other pointers indicate membership in general classes or specify particular properties. And pointers to circles representing the sides extend the depth of the description and allow more detail as shown in Fig. 5.7.

Now notice that notions like SUPPORTED-BY, ABOVE, LEFT-OF, BENEATH, and A-KIND-OF may be used not only as relations, but also as concepts. Consider SUPPORTED-BY. The statement, "The WEDGE is

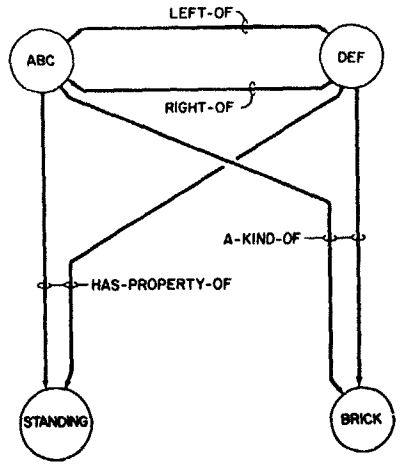


Fig. 5.6

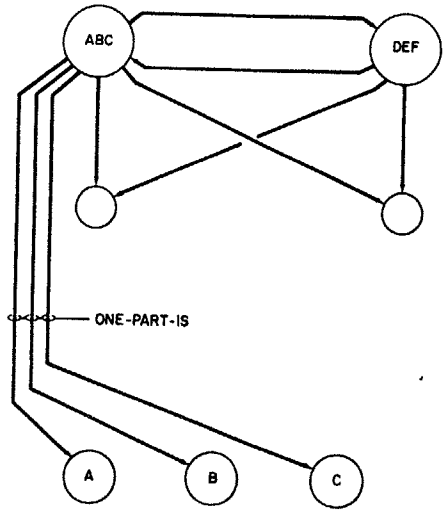


Fig. 5.7

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 5.8 shows some of the surrounding nodes of SUPPORTED-BY. I will generally call such related nodes satellites.

Thus, descriptions of relationships can be stored in a homogeneous network along with the descriptions of scenes that use those relationships.

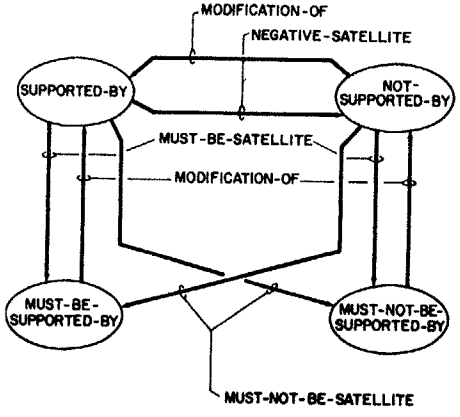


Fig. 5.8

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 functions of objects, given the appropriate network.

5.2.1 Preliminary Scene Analysis

Consider now the generation of a scene description. The starting point is a line drawing, with or 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, and HAS-PROPERTY-OF.

My system's first step in processing a scene is the application of a program written by H. N. Mahabala[1] which classifies and labels the vertexes of a scene according to the number of converging lines and the angles between them. Figure 5.9 displays the available categories. Notice that Mahabala's program finds pairs of Ts where the crossbars lie between collinear uprights.

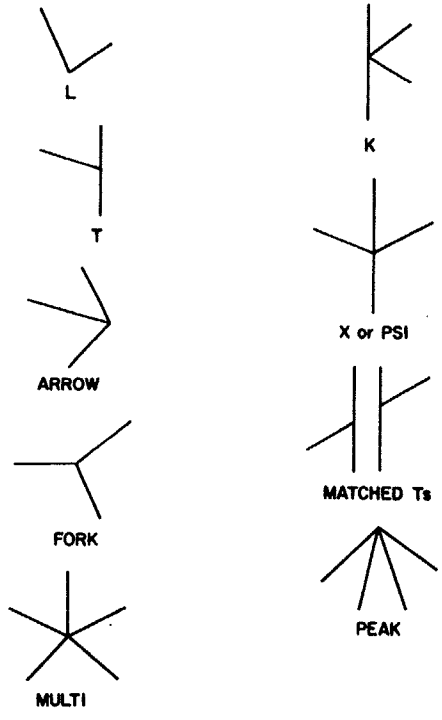


Fig. 5.9

These are called matched Ts. Such pairs occur frequently when one object partially occludes another.

Mahabala's program creates names for all of the regions in the scene. Various 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 descendants of a program developed by Adolfo Guzman.[2] This program conjectures about which regions belong to the same objects. Surprisingly it 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 regions bordering the shaft belong to the same body. This sort of evidence, together with a moderately sophisticated executive, can sort out the regions in most simple scenes.

5.2.2 Selected Relation and Property Algorithms

These programs by Guzman and Mahabala provide information required by my own description-building programs now to be described. There are programs which look for relations between objects and programs which look for properties of objects. 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, if desired.

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 5.10 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 the bottom border of the obscuring objects in Fig. 5.10. 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.

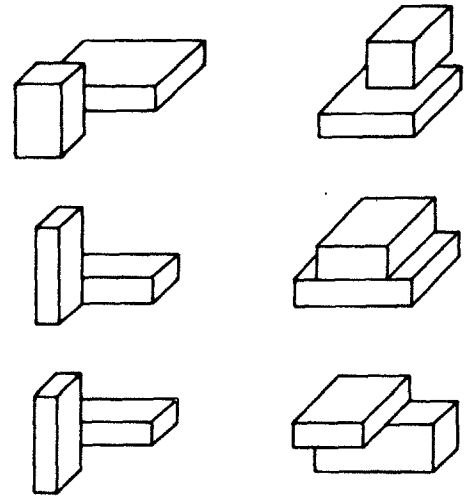


Fig. 5.10

This algorithm works on all the simple two-block situations depicted in Fig. 5.10. It even works correctly on the much more complicated, many-object scene in Fig. 5.11, 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.

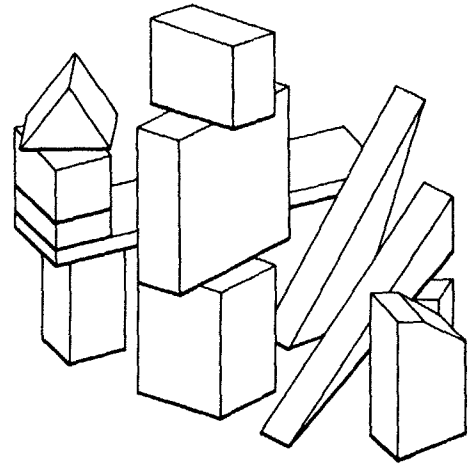


Fig. 5.11

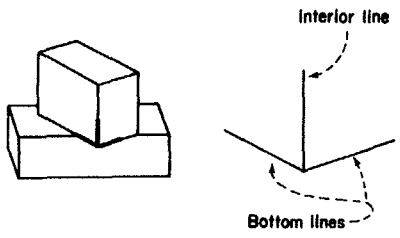


Fig. 5.12

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, psi, or a K. Then information about bottom lines is gleaned from each of the arrows, psis, 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 Fig. 5.12.)
2. If the vertex is a psi, then the two non-collinear lines are bottom line candidates. (See Fig. 5.13.)
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 Fig. 5.14.)

This is really a rule and two corollaries, rather than three separate rules. Psis and Ks result primarily when arrows appear incognito, camouflaged by an alignment of objects as illustrated by Figs. 5.13 and 14. Consequently, the corresponding rules amount to locating the arrow-forming parts of the vertex and then acting on that basic arrow.

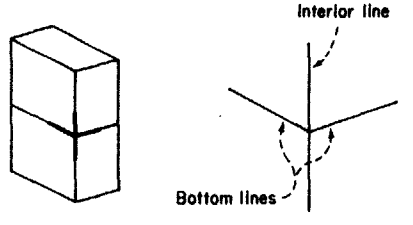


Fig. 5.13

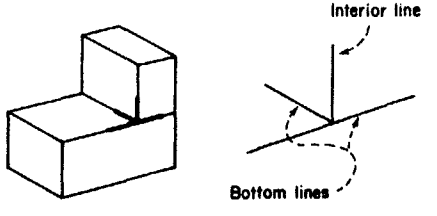


Fig. 5.14

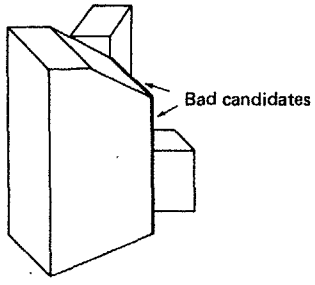


Fig. 5.15

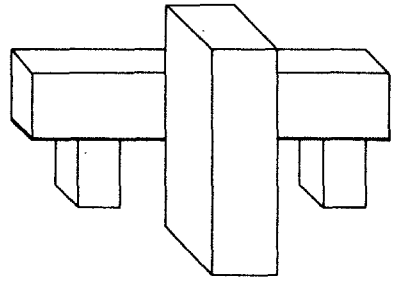


Fig. 5.16

One further step is necessary before a line can become an approved bottom line. As shown by Fig. 5.15, some of the lines which qualify 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 to 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 5.16 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.

Left and Right

Consider the spectrum of situations in Fig. 5.17. 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 of the cases I have studied. It also yields reasonable answers in Fig. 5.18 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.

Extra consideration is needed if one object extends beyond the other in both directions. No matter what the center of mass relationships, humans are reluctant to use either LEFT-OF or RIGHT-OF in such a situation. One must additionally specify a rule against this, leaving the following for LEFT-OF:

Say A is left of B $\Leftarrow\Rightarrow$

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.

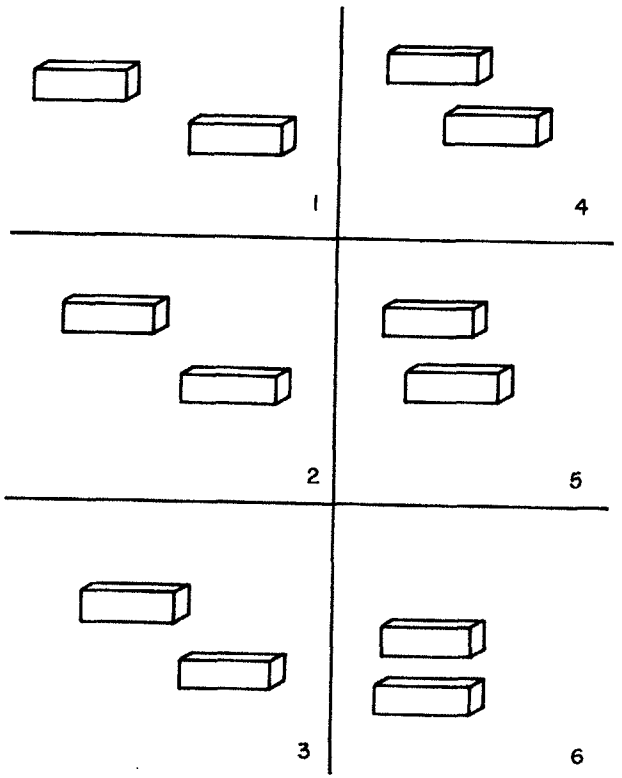


Fig. 5.17

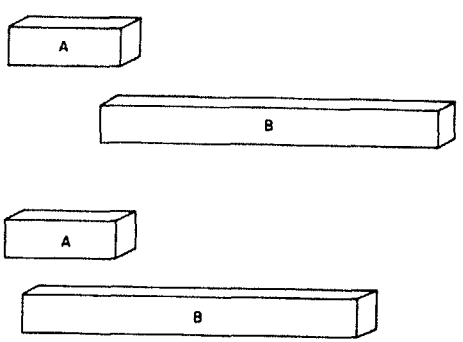


Fig. 5.18

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

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 related by IN-FRONT-OF, ON-TOP-OF, BIGGER-THAN, and so on. The orientations of objects involved are also a strong influence, and 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.

Marries

The abuts and aligned-with relations arise frequently, perhaps because of 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:

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

Thus the objects in Fig. 5.19 are said to marry one another. Those in Fig. 5.20 do not because they have no common edge. Similarly those in Fig. 5.21 do not because they have no touching faces. The MARRIES relation is sensed by methods resembling those previously described.

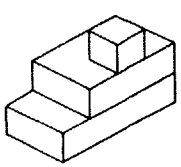


Fig. 5.19

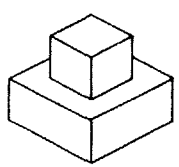


Fig. 5.20

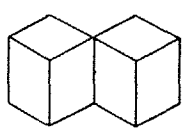


Fig. 5.21

Size

Piaget has shown that at a certain age children generally associate physical size with greatest dimension. 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 judgment. Notice that beaker A in Fig. 5.22 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 Fig. 5.23 is approximately ten times larger than object A, even if told both objects are cubes. The true factor of 27 times seems large when the trouble is taken to calculate it.

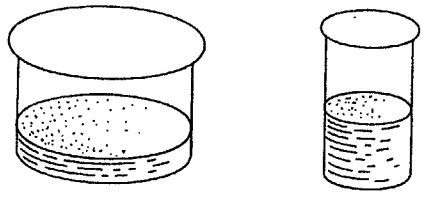


Fig. 5.22

Consequently, the size-generating program does not use 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 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

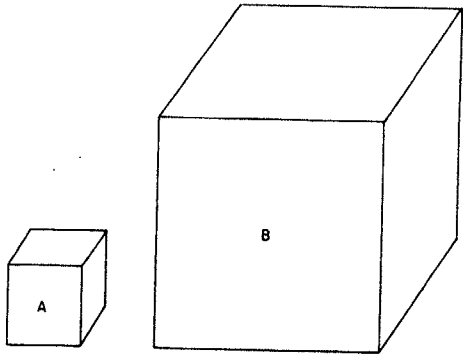


Fig. 5.23

5.3 DISCOVERING GROUPS OF OBJECTS

When a scene has more than a few objects, it is usually useful to deepen the hierarchy of the description by dividing the objects into smaller groups which can be described and thought of as individual concepts. Figure 5.24 seems to divide naturally into two groups of objects, one being three objects tied together by SUPPORTED-BY pointers, and one being three objects on top of a fourth.

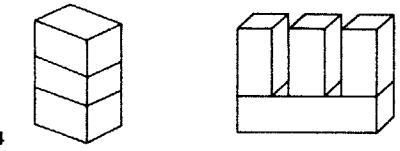


Fig. 5.24

Recognition of such groups seems to be a two part process of conjecture followed by criticism and revision. Conjectures follow from searches for objects linked by pointer chains or for objects bearing the same relation to some grouping object that binds the potential group members together. Criticism and revision is then needed to exclude from membership those objects that are weak compared with the average for the group.

5.3.1 Sequences

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

Using chains to define groups requires a rule for handling the situation illustrated by the scenes in Fig. 5.25. On the left a chain of SUPPORTED-BY pointers splits into two branches at the point where object C is supported by two objects, D and E. On the right 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 is natural to think of the scenes in Fig. 5.25 as a set of groups consisting of A-B-C, G-H-I, and J-K-L.

Another problem arises when objects tied together by a simple chain of relations should not constitute a group because of other factors. Here a need for the criticism part of the grouping process becomes clear. Figure 5.26 shows one kind of situation that can occur. In this scene the machine first conjectures 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

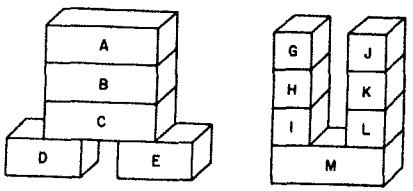


Fig. 5.25

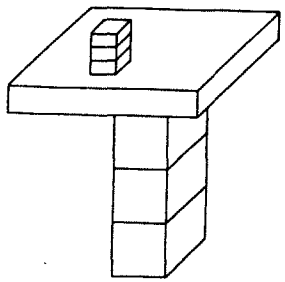


Fig. 5.26

because of the size differences and partly because of the fact that the top group is not directly over the other objects. My system uses either of these radical changes as grounds for breaking the chain.

5.3.2 Common Relations and Properties

For this kind of grouping, the basic idea is again to make a generous hypothesis as to what objects may be in a group and then to eliminate objects which seem atypical until a fairly homogeneous set remains. When several objects relate to some other object in the same way, they are immediately solid candidates for a group. The legs on the table in Fig. 5.27 are typical. They form a convincing group partly because they have the same relation to the table top and partly because all are bricks and all are standing.

All candidates for group membership must be related to one or more particular objects in the same way. For the table, all four objects are related to the board by SUPPORTED-BY. This restriction is a useful heuristic because uniform relationship to a single object seems to have strong binding power. The bricks in Fig. 5.28 naturally constitute two groups, not one.

Now it is necessary to criticize the group with a view toward finishing with a group whose members all have about the same right to group membership. Said another way, established groups where the members are very much alike should have high standards for entry while weaker groups should be more penetrable. The somewhat involved criticism algorithm now

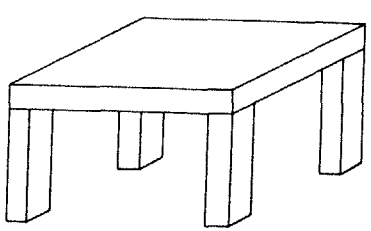


Fig. 5.27

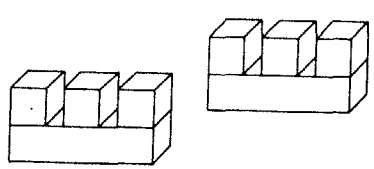


Fig. 5.28

presented helps insure this characteristic in a group by iteratively casting out the clear losers from those proposed.

The flow chart in Fig. 5.29 and the example in Fig. 5.30 help explain. The program first forms a common-relationships-lists, a list of all relationships exhibited by more than half of the candidates in the set. Objects A through F are immediately perceived to be a possible group because they all have a SUPPORTED-BY relationship with a single object G. The relationships exhibited by the candidates are:

A, B, and C:

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

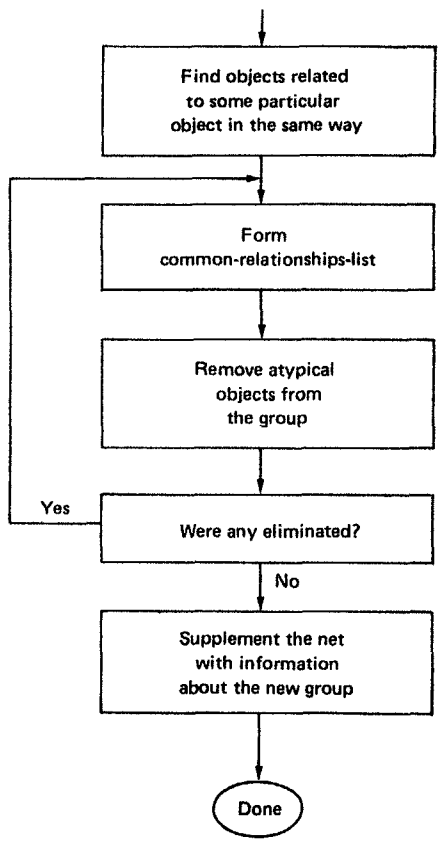


Fig. 5.29

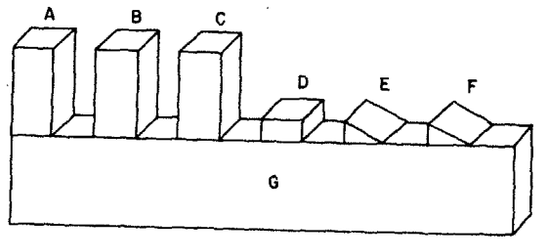


Fig. 5.30

D:

1 SUPPORTED-BY pointer to G
2 MARRIES 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 MARRIES 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:

Common-relationships-list:

1 SUPPORTED-BY pointer to G
2 MARRIES 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 shared fraction of the total number of properties in the candidate list and the common-relationships-list. 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.

Using this similarity formula to compare the various objects of the Fig. 5.30 example with the common-relationships-list, one has:

A vs. the common-relationships-list → 3/4 = .75

B vs. the common-relationships-list → 3/4 = .75

C vs. the common-relationships-list → 3/4 = .75

D vs. the common-relationships-list → 3/4 = .75

E vs. the common-relationships-list → 2/5 = .20

F vs. 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 percent 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 until 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 MARRIES 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 vs. the common-relationships-list → 4/4 = 1

B vs. the common-relationships-list → 4/4 = 1

C vs. the common-relationships-list → 4/4 = 1

D vs. 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 alike.

5.3.3 Other Kinds of Grouping

There obviously cannot be a single universal grouping 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

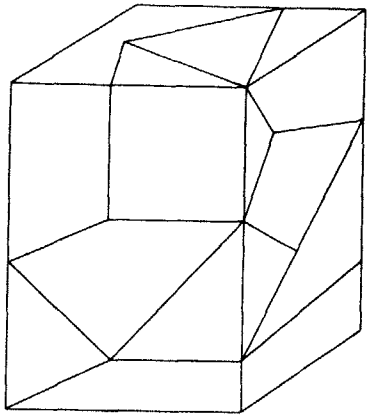


Fig. 5.31

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 in this area are again largely unexplored, but it seems that overall shape can sometimes impose unity on a complete hodge-podge. Figure 5.31 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 neighbors.

5.3.4 Describing a Group Using 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 5.32 illustrates this step for a group of three objects, A, B, and C, all arranged in a tower.

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. For our A B C case, the typical member is described as a kind of

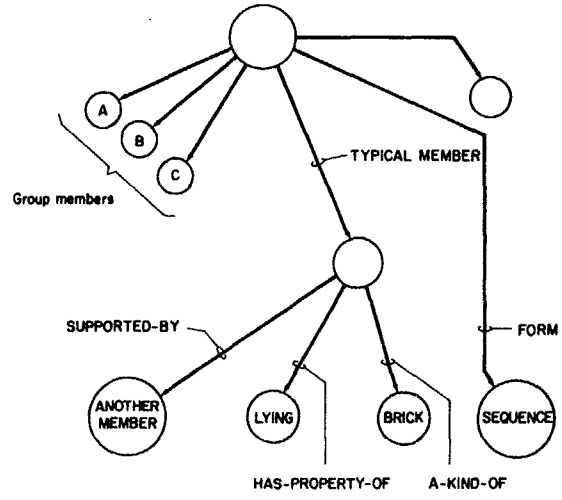


Fig. 5.32

brick, as lying, and as on top of another member of the group. Notice also the FORM pointer to SEQUENCE which indicates the kind of group formed.

5.4 NETWORK SIMILARITIES AND DIFFERENCES

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 5.33 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 Fig. 5.33 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 similar nevertheless.

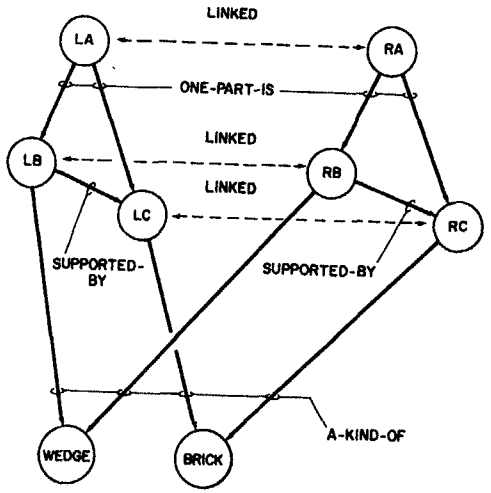


Fig. 5.33

5.4.1 The Skeleton and Comparison Notes

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 Fig. 5.34 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.

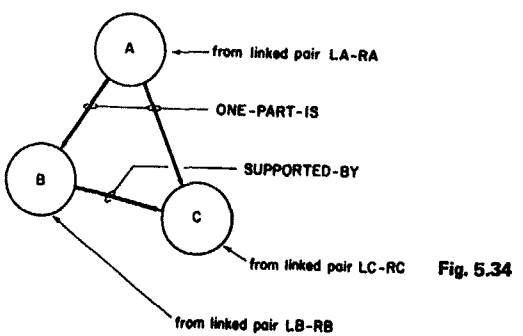


Fig. 5.34

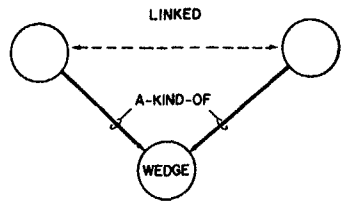


Fig. 5.35

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 comparison note or C-NOTE for short. The most common type of comparison note is the intersection comparison 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 corresponding objects from two scenes are both wedges. Then both concepts exhibit an A-KIND-OF pointer to the concept WEDGE as shown by Fig. 5.35. 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 5.36 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 comparison notes look like this intersection paradigm.

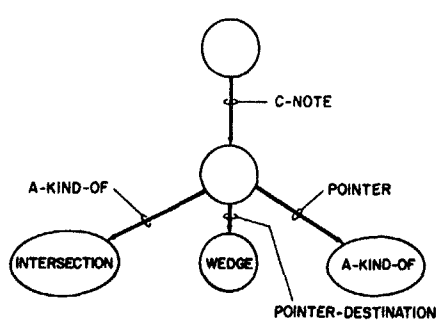


Fig. 5.36

5.4.2 Evans' Analogy 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,[3] can understand why this is a powerful feature, rather 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 5.37 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.

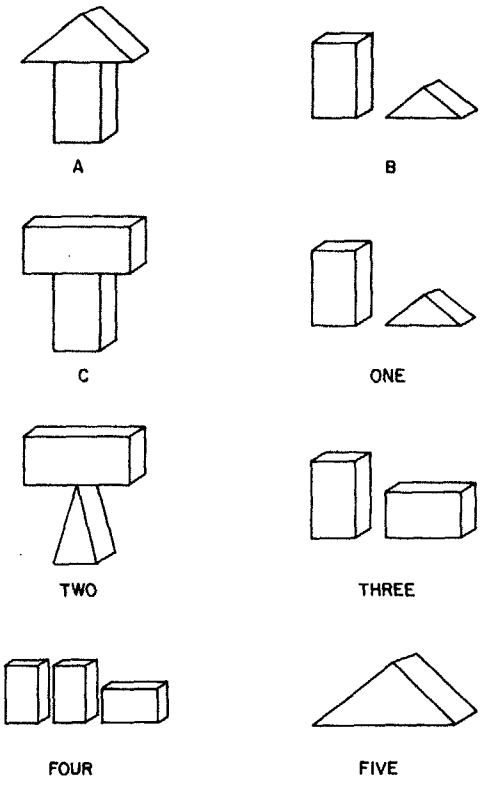


Fig. 5.37

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 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

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 that serves to identify some scene with some member of a group of models. It works because the identification problem entirely parallels the problem of identifying a given A to B transformation description with some member of the group of answer connected C to X transformations. The identification program, together with a short executive routine, handles the problem of Fig. 5.37 easily, correctly reporting scene three as the best answer. Reasonably enough, the machine thinks scene one is the second 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 these 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 is superior to mine in this respect because it can often compare two drawings in more than one way. It can visualize some changes as either reflections or any 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 descripiton, 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.

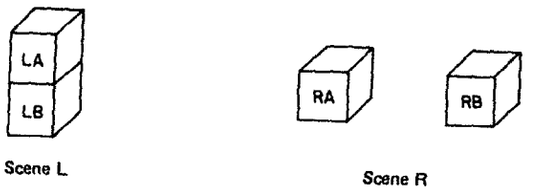


Fig. 5.38

5.4.3 A Catalog of Comparison-Note Types

The Supplementary-Pointer and the Exit

Consider the scenes in Fig. 5.38 and their descriptions in Fig. 5.39. 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 comparison note and has the form shown in Fig. 5.40.

Suppose now we consider a standing brick and compare it with a cube. Here the linked concepts would differ only in that the brick has an additional pointer identifying it as standing. This differs from the supplementary-pointer case in that STANDING is a node outside the scene description. A pointer to the concept EXIT signals this situation. 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.

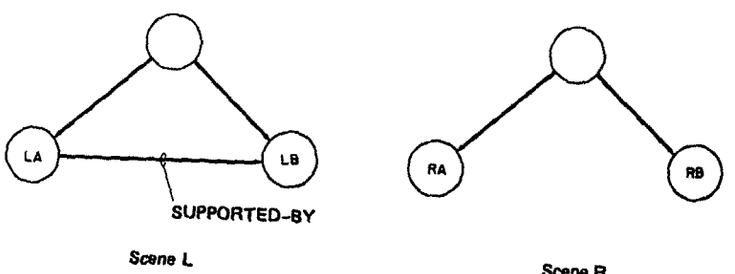


Fig. 5.39

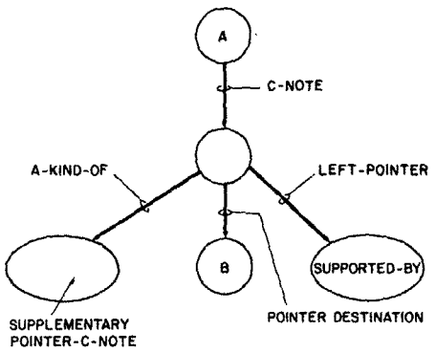


Fig. 5.40

Pointer Modifications

Suppose the left and right networks in Fig. 5.41 are compared. Notice the MARRIES pointer between LA and LB and the DOES-NOT-MARRY pointer between RA and RB. These could be handled individually as unrelated supplementary-pointer comparison notes, but this would ignore the close relationship between MARRIES and DOES-NOT-MARRY. Consequently a different type of comparison note is used that recognizes the relationship. It is the negative-satellite-pair comparison note. With it, the comparison looks as shown in Fig. 5.42. To find such negative-satellite-pair comparison notes, the comparison programs peruse the descriptions of unmatched pointers between linked pairs for evidence of relationship. For example, MARRIES 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 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

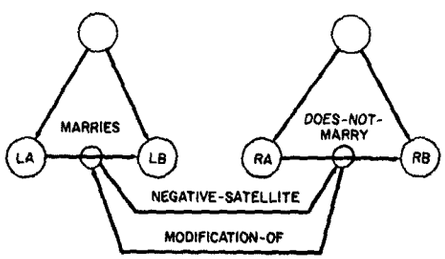


Fig. 5.41

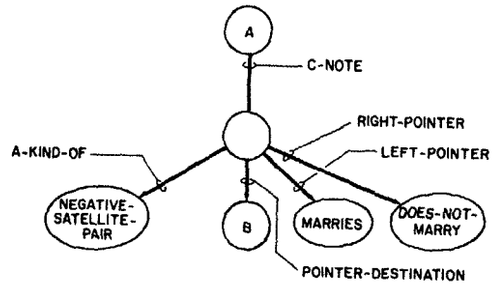


Fig. 5.42

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.

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

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 Fig. 5.43. This is very much like the pointer-satellite idea with A-KIND-OF replacing MODIFICATION-OF. In any case, the description generator recognizes this and similar situations and again generates a group of comparison note types. The first of these is the A-KIND-OF chain illustrated by the above situation. This causes the comparison note of Fig. 5.44.

The a-kind-of-chain comparison 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

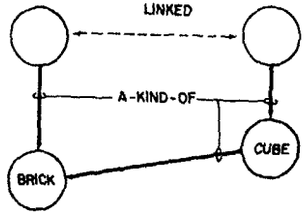


Fig. 5.43

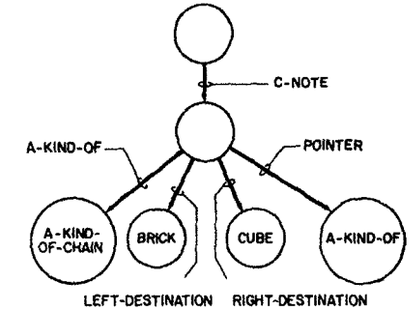


Fig. 5.44

object for which no identification can be made. There is still an a-kind-of-chain comparison note because CUBE is linked to the general concept OBJECT by a sequence of A-KIND-OF relations.

Another kind of popular concept modification is the a-kind-of-merge comparison note. These a-kind-of-merge comparison 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.

5.5 LEARNING AND THE NEAR MISS

I can now discuss the problem of learning to recognize simple block configurations. Although this may seem like a very special kind of learning, I think the implications are far ranging, because I believe 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 of learning there is frequently an unstated assumption 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.

To begin with I want to make clear a 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 Fig. 5.45.

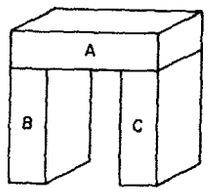


Fig. 5.45

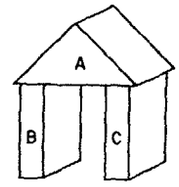


Fig. 5.46

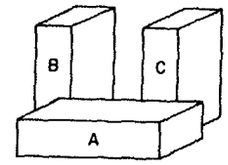


Fig. 5.47

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

Now suppose the description containing these facts are compared with the scene in Fig. 5.46, where object A is a wedge, and with the scene in Fig. 5.47, 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 equally likely if 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 Fig. 5.46 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 Fig. 5.47 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 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.

In the learning of such models, near misses are the really important learning samples. 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. 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 inadequate 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.

5.5.1 Elementary Model Building Operations

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. One has a sequence of more and more sophisticated models.

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. Later I discuss in more detail how the alternative branches occur in the model development sequence. 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 tables at the end of this section summarize the results.

The A-Kind-of-Merge: Example Case

Suppose the initial model consists of a plain brick while the example is a wedge. Figure 5.48 shows the resulting comparison description. Only one difference is found: the object of the model is related to BRICK while the object of the example is related to WEDGE. But since both BRICK and WEDGE relate by A-KIND-OF to OBJECT, the a-kind-of-merge comparison note occurs. Several explanations and companion responses are possible. One is that the source of the comparison 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 two such classes are simply the concepts OBJECT and RIGHT-PRISM. These are both located as the intersection of A-KIND-OF paths. The program responds by replacing the pointer in the comparison network that points to the a-kind-of-merge comparison note by an A-KIND-OF pointer to one of the intersection or merge concepts. In this case an A-KIND-OF pointer is installed between the comparison note origin and the concept OBJECT. Here the altered comparison network is the new model shown in Fig. 5.49. Note that this primary response I have selected for the machine represents a moderate stand with respect to a rather serious induction problem. I have avoided the extremes of pointing to

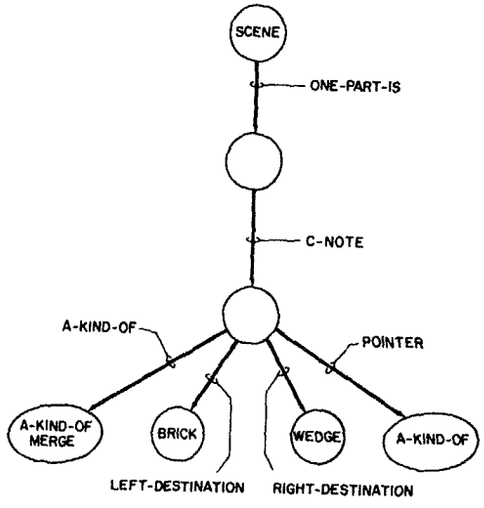


Fig. 5.48

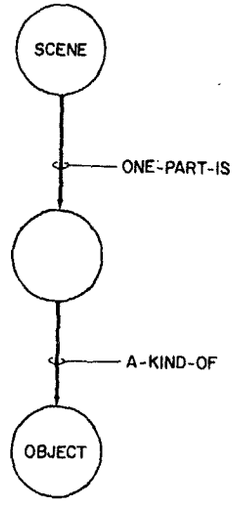
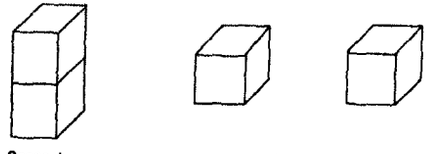


Fig. 5.49

THING or the OR of brick and wedge, but just where in the spectrum to settle on is a difficult question. Another reasonable position would be to choose RIGHT-PRISM, for example.

The Supplementary-Pointer: Near Miss Case

Now suppose Scene 1 in Fig. 5.50 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 comparison 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 comparison note turns up, it transforms to the emphatic MUST version of the pointer involved. Thus the new model is the one in Fig. 5.51.



Scene 1 Scene 2

Fig. 5.50

Of course the supplementary pointer can turn up in the near miss as well as in the current model. Suppose Scene 1 in Fig. 5.50 is the near miss

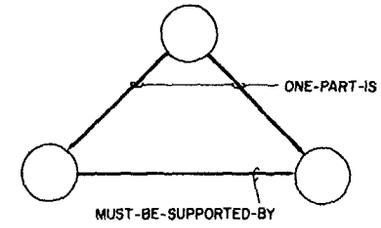


Fig. 5.51

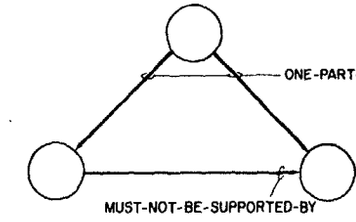


Fig. 5.52

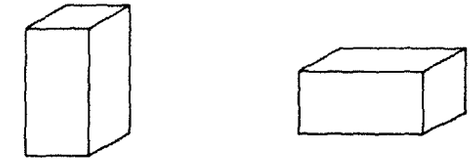
instead of the current model. One concludes A cannot be on B. The supplementary-pointer comparison note now indicates a relation that apparently cannot hold. Appropriately, the MUST-NOT version of the supplementary pointer is substituted in and the new net appears as in Fig. 5.52.

The Must-Satellite-Pair

Frequently comparison between the current model and a new sample displays comparison notes that do not reveal any new feature, but rather result from 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 MARRIES pointer. Now clearly the MARRIES pointer is appropriate in the description and the must-satellite-pair comparison 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.

The A-Kind-of-Merge: Near Miss Case

Sometimes a comparison note offers two or more nearly equal explanations. Consider the very simple current model and near miss in Fig. 5.53. The comparison 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



Current model Near miss

Fig. 5.53

TABLE 5.1 Action of concept generator: Example case

Comparison note type	Pointer involved	Response
A-kind-of-chain	–	Point to intersection with model's pointer
A-kind-of-merge	–	Point to intersection with model's pointer
		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

not standing. Responding to these explanations, the model builder might replace the a-kind-of-merge comparison 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 antiproperties, the MUST version is considered more likely. (Tables 5.1 and 5.2 summarize the points made in this section.)

5.5.2 Coping with Multiple Differences

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

The model builder must therefore consider all the comparison 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 hypothesis 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

TABLE 5.2 Action of concept generator: Near miss case

Comparison note type	Pointer involved	Response
A-kind-of-chain	–	If model's node is at the end of the chain add must-not-be satellite to near miss' node
		If near miss' node is at the end of the chain, use must-be satellite to model's node
A-kind-of-merge	–	Replace model's pointer by its must-be satellite
		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-satellite pair	–	Retain model's pointer
Supplementary-pointer or exit	Fundamental pointer in the model	Replace pointer with its must-be satellite
	Fundamental pointer in the near miss	Insert pointer into the model 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

comparison 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 comparison notes had but one interpretation, only one new branch would be generated. The a-kind-of-merge comparison note has two possible interpretations, however, and if one such comparison note occurs, it is only reasonable to create two branches instead of one. The action on the other comparison notes is the same for both 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 by simply 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 most obvious way to search for key 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 comparison notes which are nearest to the origin of the comparison description. All those candidates found at greater depth are considered secondary.

The highest level differences allow quick formation of little hypotheses about why the near miss misses and what to do as a consequence. A complete hypothesis specifies one comparison note as the sole cause of the miss and it further specifies which interpretation of that comparison note is assumed. Consequently there is a hypothesis for each interpretation of each potentially central comparison note.

The comparison note specified as crucial by a hypothesis is transformed as if it were the only comparison note. The other comparison notes 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 comparison notes but one are treated exactly as if the near miss were not a miss at all!

So far a single comparison 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. I have therefore decided that only one special combination of two comparison notes is ever permitted to form a hypothesis.

Hypotheses based on two contributing comparison notes are added to the hypothesis list only when two comparison notes with nearly identical descriptions occur.

Current model Near miss

Fig. 5.54

Consider Fig. 5.54. 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 comparison notes in the miss-explanation mode.

5.5.3 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 the very simple situation already presented back in Fig. 5.53. Think again of the left side as the current model and the right side as the near miss. The current model and the near miss combination generate an a-kind-of-merge comparison note for which the priority interpretation is that examples of the concept must 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 a tilted brick does the job. A 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 to select another alternative.

In the case at hand, an alternative is found and the must-not-be-lying interpretation of the comparison between the scenes leads to a new intermediate model. This in turn is refined by the tilted brick scene 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.

5.6 SOME GENERATED CONCEPTS

In this section 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 or eating, so the programs discussed here handle only some of the physical aspects of these human notions. 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 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.

5.6.1 The House

Figure 5.55(*a*) 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 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 Fig. 5.55(*a*). Then its description generation apparatus contributes the network which serves as the first unrefined, unembellished model of Fig. 5.56. Now suppose the scene in Fig. 5.55(*b*), a near miss, is the next sample. Its net is that shown in Fig. 5.57. The only difference is the supplementary pointer SUPPORTED-BY. Glancing at Table 5.2, it is clear that the overall result is conversion of the

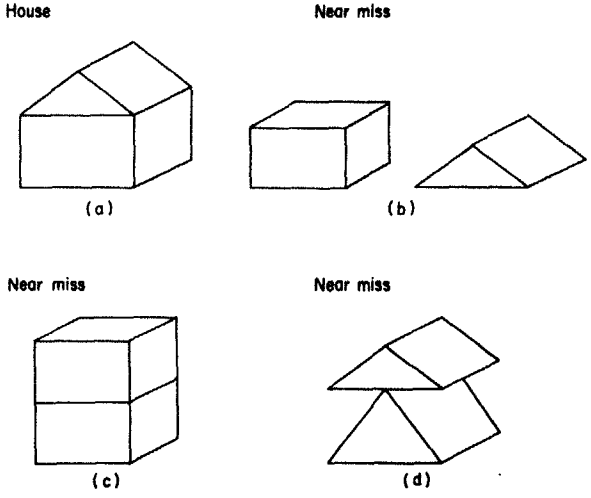


Fig. 5.55

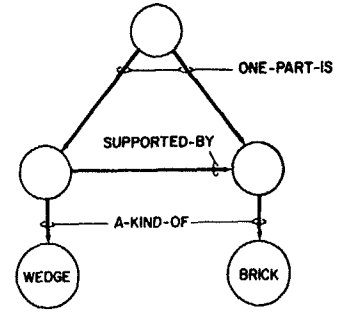


Fig. 5.56

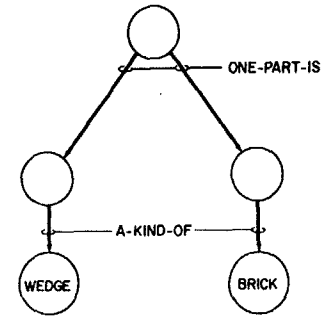


Fig. 5.57

SUPPORTED-BY pointer in the old model to MUST-BE-SUPPORTED-BY in the new model. Thus the new model is that of Fig. 5.58.

Much is yet to be learned. For one thing, the top object certainly must be a wedge. Showing the machine the near miss of Fig. 5.55(*c*) conveys this point immediately. Similarly the near miss of Fig. 5.55(*d*) makes the brick property of the bottom object mandatory. But notice that 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 shown in Fig. 5.59.

5.6.2 The Tent

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

The base model is the description of the scene in Fig. 5.60(*a*) and the first sample is the near miss in Fig. 5.60(*b*). Two a-kind-of-merge comparison

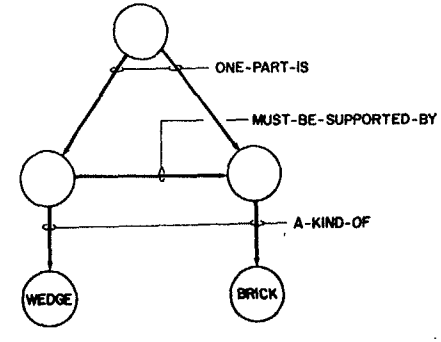


Fig. 5.58

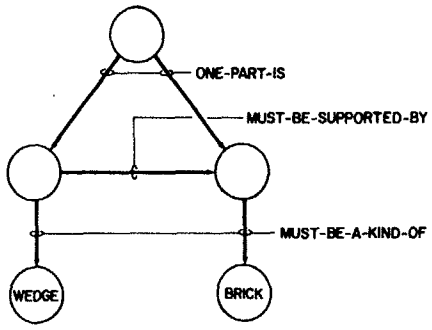


Fig. 5.59

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 has priority. Now this result is complemented by the near miss in Fig. 5.60(*c*) which informs the machine of the importance of the MARRIES relation. Again dual comparison notes announce the loss of a pair of MARRIES pointers, and twin MUST-MARRY pointers are installed.

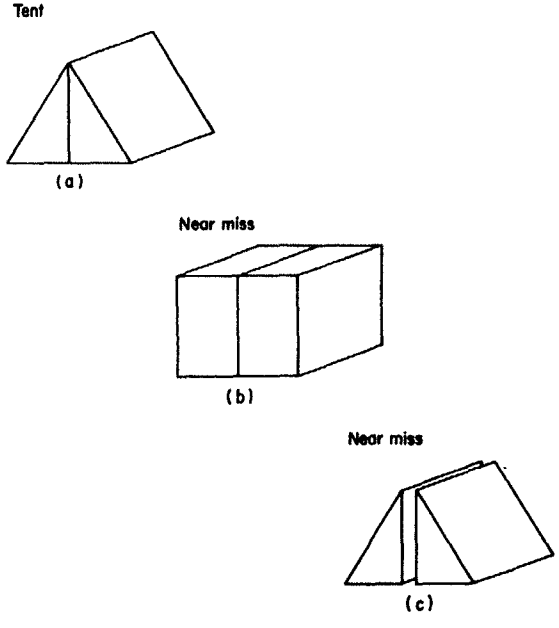


Fig. 5.60

5.6.3 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 model tree and a challenge to routines that select priority hypotheses.

An arch with sides neatly aligned with the lintel forms the first model. Combining this with the scene in Fig. 5.61(*a*) the machine deduces that the MARRIES relations between the top and the supports are not crucial.

Next the near miss of Fig. 5.61(*b*) 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 Fig. 5.61(*c*). Here the two supports touch, supplying two MARRIES 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 to be 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.

Finally, the top object is not necessarily a brick. The sample in Fig. 5.61(*d*) 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.

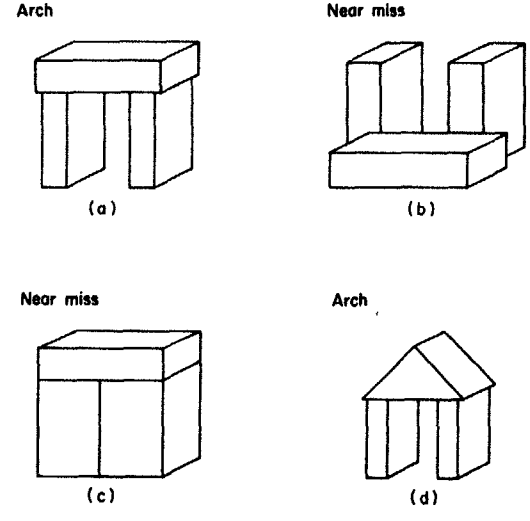


Fig. 5.61

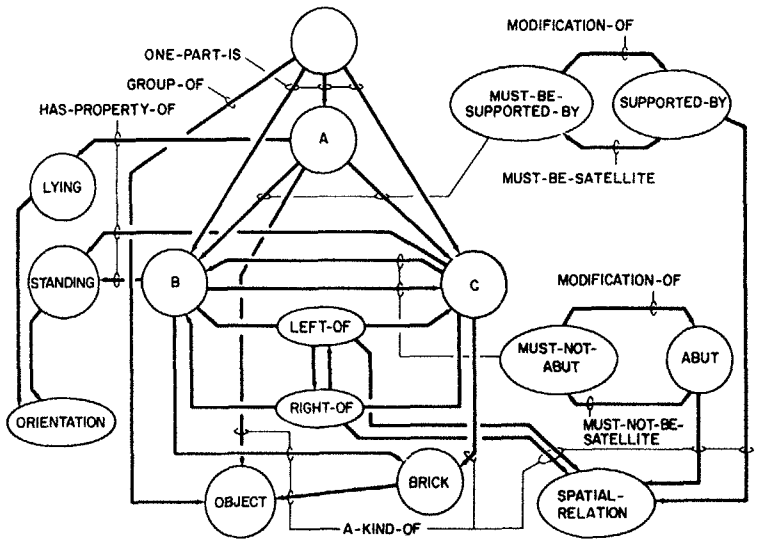


Fig. 5.62

Figure 5.62 shows the resulting model. I give it in somewhat more detail than usual to convey a feeling for the complexity the programs actually deal with.

5.6.4 The Table

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

Study the table in Fig. 5.63 and the description in Fig. 5.64. 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 near miss non-table of Fig. 5.65(*a*). Moreover, this conception of table excludes structures such as that in Fig. 5.65(*b*), a fact which is handily incorporated through a MUST-NOT-MARRY pointer. Next, since the non-

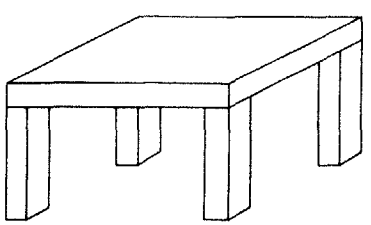


Fig. 5.63

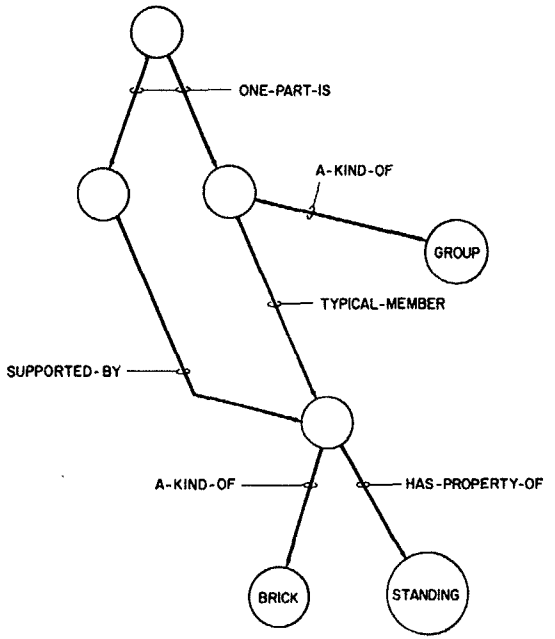


Fig. 5.64

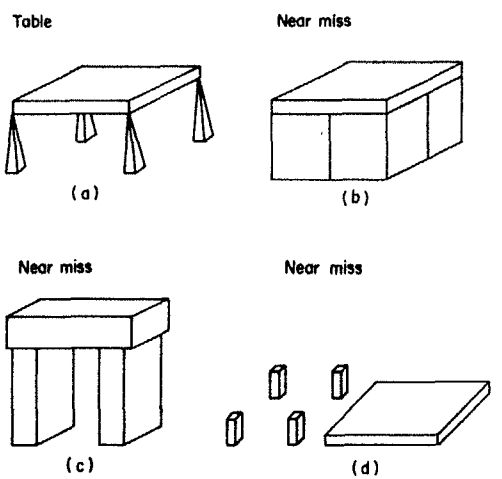


Fig. 5.65

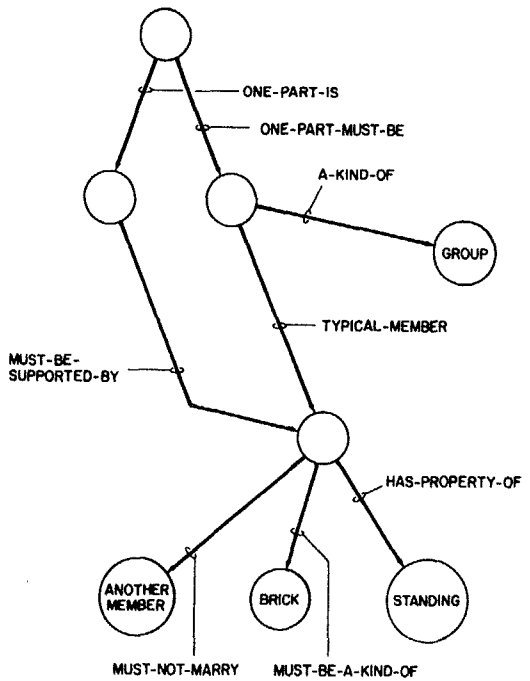


Fig. 5.66

table in Fig. 5.65(*c*) has only two supports, no grouping occurs, which leads to insistence on a group in the next model refinement. Finally, the scene in Fig. 5.65(*d*) leads to replacement of the SUPPORTED-BY pointer by MUST-BE-SUPPORTED-BY. Figure 5.66 shows the last model in this development.

5.7 IDENTIFICATION

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 section have the following kind of responsibilities:

1. To see if two scenes are identical.
2. To compare some scene with a list of models and report the most acceptable match. This is the identification problem in its simplest form.
3. To identify some particular object 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.
4. 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.

5.7.1 Exact Match and Discovering Symmetry

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 comparison notes that describe how the parts of the scene are anchored to the general store of concepts. None of the other types of comparison notes appear because identical scenes cannot produce two networks with the necessary aberrations of form.

Conversely, if comparison of two networks results in intersection comparison 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 Fig. 5.67 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. The scenes are clearly not identical with respect to a program with such a capability.

It is interesting to note in passing that 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:

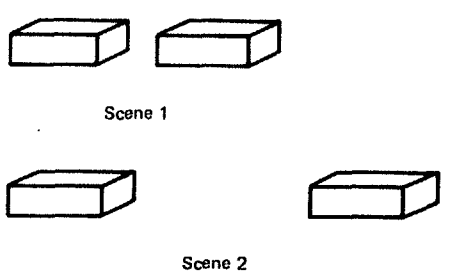


Fig. 5.67

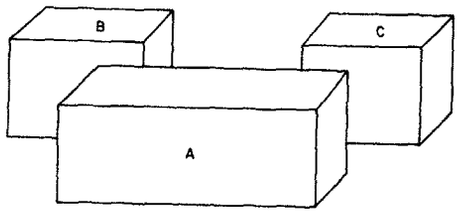


Fig. 5.68

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 analog of x-coordinate negation and network matching corresponds to a check for invariance.

To see how this works, consider the scene in Fig. 5.68. The center object A is flanked by B on the left and by C on the right. Figure 5.69 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 5.70 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 A′, B with C′, and C with B′. Since there are no differences, the machine concludes the scene is in fact symmetric.

The machine knows LEFT-OF and RIGHT-OF are opposites because they are linked together by OPPOSITE pointers. Consequently, it is

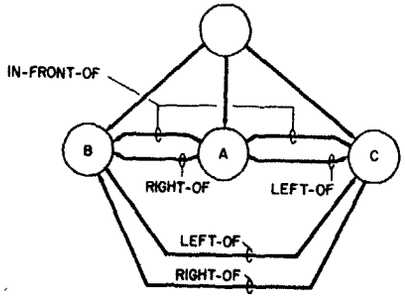


Fig. 5.69

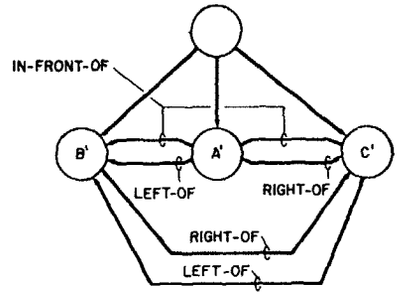


Fig. 5.70

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 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.

An interesting combination is a simultaneous LEFT-RIGHT and an IN-FRONT-OF–BEHIND SWITCH. This one gives the machine a chance of realizing that two scenes are simply front and back views of the same configuration as are the scenes in Fig. 5.71.

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 comparison notes from the local ones already at hand, thereby introducing the kind of hierarchy into the comparison descriptions that is already the standard in scene descriptions. One obvious ability of such a program would be that of noticing a preponderance of similar comparison notes. This and some of the double comparison ideas proven useful in doing analogy problems are the things the machine needs to learn about symmetry.

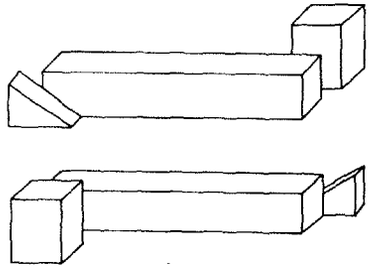


Fig. 5.71

5.7.2 Best Match for Isolated Structures

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 comparison 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 comparison 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 comparison 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 Fig. 5.72 cannot be identified as an arch because the network describing the near-arch has MARRIES 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 comparison note that positively prevents a match.

Identification with a particular model is also rejected if the difference description contains exits or supplementary-pointer comparison notes which involve must-be satellites. Such comparison notes occur when essential relations or properties are missing in the unknown. Two bricks lying on a table do not form a pedestal because the model for the pedestal has a MUST-BE-SUPPORTED-BY pointer. The result is a supplementary-pointer comparison note involving the must-be satellite MUST-BE-SUPPORTED-BY. Again there is no match.

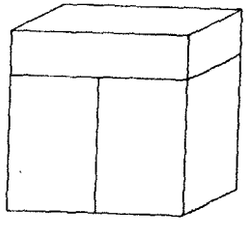


Fig. 5.72

Suppose we have a HOUSE but its identity is as yet unknown. Match of a HOUSE against the PEDESTAL, the TENT, and the ARCH all lead to difference descriptions with comparison 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 HOUSE. This in turn causes a fatal exit comparison note in the difference description.

The next problem emerges because some unknown may acceptably match more than one model in a trail list. 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 results by forming a weighted sum for each comparison. This seems to work well enough for the moment, but I do not think it would pay to put much effort into tuning such a formula. Instead more knowledge about the priorities of differences should lead to far better programs that do not use such a primitive scoring mechanism.

5.7.3 Best Match for Structures in a Context

Examine Fig. 5.73. 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 models and identification process of the system as described so far, because so far anything identified as a 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 Fig. 5.73.

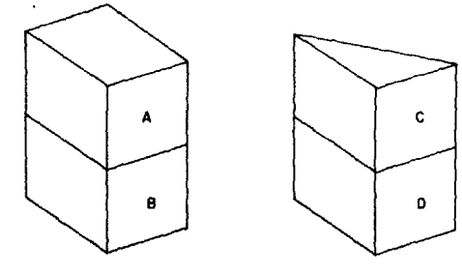


Fig. 5.73

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.

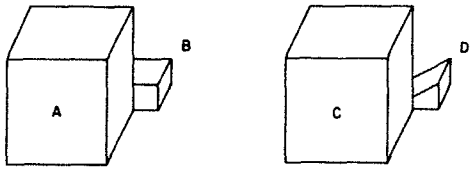


Fig. 5.74

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 best matching model, but only to verify that a particular identification is not contradictory. This means that object B in Fig. 5.74 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.

5.7.4 Learning from Mistakes

Suppose the program attempts to identify a house as a pedestal. Identification fails because the wedge will not match the top of the pedestal and the resulting type of a-kind-of-merge comparison 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 armamentarium 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.

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

description of 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, an unknown which happened to be a house 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. Notice that the process requires the same steps as do analogy problems as described earlier. Figure 5.75 clarifies the procedure.

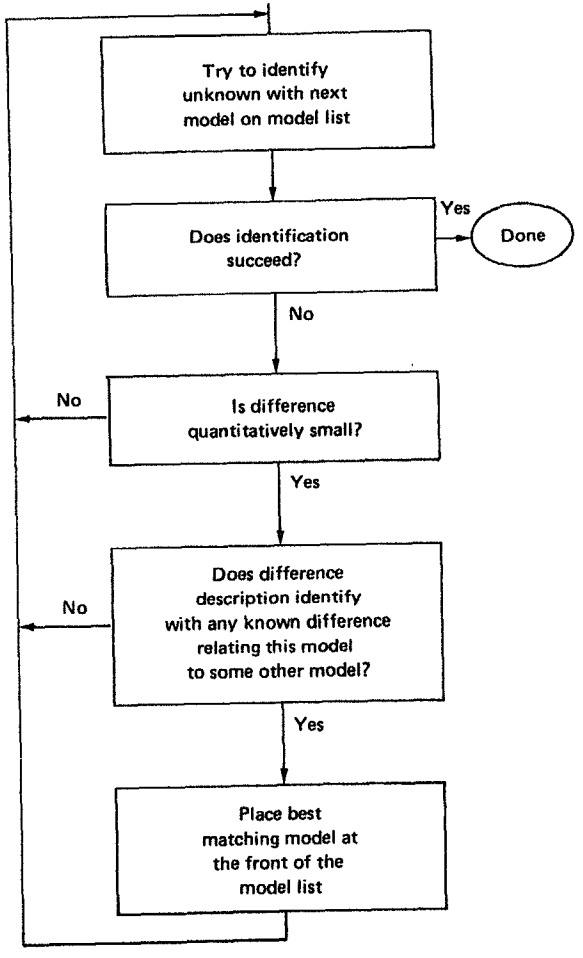


Fig. 5.75

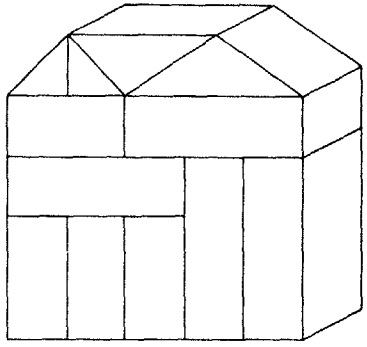


Fig. 5.76

5.7.5 Finding a Needle in a Haystack

The scene of Fig. 5.76 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. The result is a set of links between the objects of the model and their nearest analogues in the scene.
2. Once a good group of objects is picked, then the pointers relating these objects to the other objects in the scene are temporarily forgotten. In human terms, this is like painting the subgroup a special color.
3. Finally, with the best group of objects set into relief by the previous excision, the ordinary identification routines are applied with the expectation of reasonable performance.

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

REFERENCES

1. Mahabala, H. N. V.: Preprocessor for Programs which Recognize Scenes, *M.I.T. Artificial Intelligence Laboratory Memo* 177, 1969.
2. Guzman, Adolfo: "Computer Recognition of Three-dimensional Objects in a Visual Scene," Ph.D. thesis, MAC-TR-59, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., 1968.
3. Evans, Thomas G.: "A Heuristic Program to Solve Geometric Analogy Problems," Ph.D. thesis, in Marvin Minsky (ed.), "Semantic Information Processing," The M.I.T. Press, Cambridge, Mass., 1963.