Technical Report 283

A Planning System For Robot Construction Tasks

Scott E. Fahlman

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

A PLANNING SYSTEM FOR ROBOT CONSTRUCTION TASKS

Scott E. Fahlman

May 1973

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY
Cambridge, Massachusetts 02139

ABSTRACT

This paper describes BUILD, a computer program which generates plans for building specified structures out of simple objects such as toy blocks. A powerful heuristic control structure enables BUILD to use a number of sophisticated construction techniques in its plans. Among these are the incorporation of pre-existing structure into the final design, pre-assembly of movable sub-structures on the table, and the use of extra blocks as temporary supports and counterweights in the course of the construction.

BUILD does its planning in a modeled 3-space in which blocks of various shapes and sizes can be represented in any orientation and location. The modeling system can maintain several world models at once, and contains modules for displaying states, testing them for inter-object contact and collision, and for checking the stability of complex structures involving frictional forces.

Various alternative approaches are discussed, and suggestions are included for the extension of BUILD-like systems to other domains. Also discussed are the merits of BUILD's implementation language, CONNIVER, for this type of problem solving.

ACKNOWLEDGEMENTS*

I would like to express my sincere gratitude to the many members of the Artificial Intelligence Laboratory whose direct and indirect support made this work possible. Particular thanks must go to Patrick Winston for suggesting this topic originally and for patiently advising my research, even though the pace was often maddeningly slow; to Terry Winograd for performing the initial reconnaissance of this problem area; and to Gerry Sussman, Drew McDermott, Carl Hewitt and Jon White who provided many good ideas and helped me slay many bugs. Finally, I would like to express my appreciation to my wife, Penny, for helping with the typing and diagrams, but mostly for putting up with it all.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-70-A-0362-0005.

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Science, June 1973.

TABLE OF CONTENTS

Chapter 1: Introduction

1.1 The Role of Problem Solving in a Robot

One of the goals of artificial intelligence research is the creation of robots, artificial slaves that will do man's bidding and relieve him of any tasks that he finds dangerous, distasteful or uninteresting. At present, machines exist with muscular and computational powers far in excess of human capabilities but, except in the most standardized and predictable tasks, they require constant human supervision and control. Part of the problem results from the machines' inadequate means of sensing their environment; until an effective computer vision system is developed, men will continue to be depended upon for their eyes. Equally important, however, is the machines' weakness and dependency in the area of planning and problem solving. Somehow, given a goal, a situation, and a set of constraints that must be met, a plan must be generated to accomplish that goal. This plan-generating process is the subject of this thesis.

If a robot is to deal effectively with a wide variety of tasks and situations, it will need to know a number of techniques, procedures, and tricks, and will have to be able

to choose and apply these techniques at appropriate times. Some of these techniques will be quite general and can be used to solve whole classes of problems in widely varying sets of circumstances; others will be useful only in a few particular cases, but could result in large savings of effort or in success where the more general methods failed. In order to choose an appropriate technique, then, it will be necessary for the robot to be able to recognize and classify the problems that confront it as it works on its assigned tasks.

Even with a good technique selector, sometimes the first method tried will fail. Similarly, an arbitrary decision will sometimes have to be made and this may later prove to have been the wrong choice. When the failure is discovered the robot will need to undo any damage it has done to its own data bases (and perhaps to its external environment), while at the same time preserving any information it has discovered which might aid it in making the next choice or perhaps in accomplishing some other pending goal.

It is clearly not desirable to have this groping and blundering occur in the real physical world. Questions of efficiency aside, a simple failure could result in a gross alteration of the robot's environment or anatomy, or, at the

very least, destruction of some structure that has just been laboriously built. For this reason it is essential that the robot have some mental model of the world in which it is working. Using this model the robot can test in advance any step or series of steps in its plan and get what is hopefully a good indication of the results. Sometimes, of course, a discrepancy will occur due to inaccuracy or over-simplification in the model, but these events will be far less common than would be the case with no model. Ideally, the robot could discover patterns in these discrepancies, and improve its model accordingly.

Except in very special cases it is not worth the extra computation to produce a truly optimal plan, but the plan should not be blatantly stupid by human standards. For example, if the robot were told to move a thousand small parts across the room, it should find a container to carry them in, not make a thousand separate trips. On the other hand, we would not complain too much if it chose a route that was a couple of steps too long or if its hand did not follow the best traveling-salesman route in picking up the objects. Similarly, it is less important that the robot do well on very hard problems than that it consistently succeed in overcoming the simple problems that it will face far more often. A useful robot need not be a genius, but its

programs must be organized in such a way that they do not exhibit the typical collapse when some combination of simple circumstances leads to a program bug. Other techniques should be tried or, if all else fails, the robot should realize that it is losing and call for help. Again ideally, if the robot is shown a way to accomplish some task that is better than its own plan, we would like it to figure out where it went wrong and alter its own programs accordingly, or to generalize the better plan and store it away as a new technique.

1.2 The BUILD System: Goals, Abilities, and Limitations

It would of course be impossible to attack the entire area of robot problem solving at once. Therefore, I have limited my own investigation to the set of problems that a one-handed robot would encounter while attempting to build specified structures out of simple objects such as toy blocks. The world of blocks is ideal for this type of study because it provides difficult and interesting problems, but nevertheless is very simple and self-contained. Since the robot needs to know only a few concepts about gravity, support and friction, it is possible in such a world to study the organization of planning programs without facing

the difficulties of collecting, maintaining, and effectively using a huge body of real-world knowledge. Games such as chess share this closure property, but are farther removed from the type of useful real-world activity which we would like the robot eventually to perform.

In the course of this investigation I have written a set of programs, collectively called BUILD, which, by operating on internal 3-dimensional models, produces a plan for converting some present state of a table full of blocks into some desired or goal state. The present state is given to the system in the form of a complete 3-dimensional model indicating the size, shape, position, and rotation of each block in the scene. Such a model could be input directly by a human operator, or it could be produced by another program such as a vision system looking at the table or a language system working from a verbal description. The goal state is presented in an identical format, except that it may be incomplete. Any block which appears in the present state but not in the goal state is assumed to be unimportant in the final design; BUILD is free to put such a block anywhere, as long as it ends up out of the way of the specified structures. The plan produced consists of a list structure containing, in their proper sequence, all of the block movements that are to be made, along with information

as to why each move was made at that particular time. This goal information is saved because it will be useful to other programs which compare or modify plans.

Though much of the necessary groundwork has been laid, BUILD does not at present carry its planning down to the level of actual hand movements and finding paths through 3-space. In most cases this is a fairly trivial process, but a good general solution would have to deal with several difficult problems that are not yet solved. One of these is the development of a good way to model empty space, for the purpose of efficient path-finding. It has been my feeling that since hand motion can be neatly isolated from the remainder of construction planning, I could better use the available time by concentrating in the other areas. BUILD therefore operates as if the blocks could be moved by magic, disappearing from one position and reappearing in another.

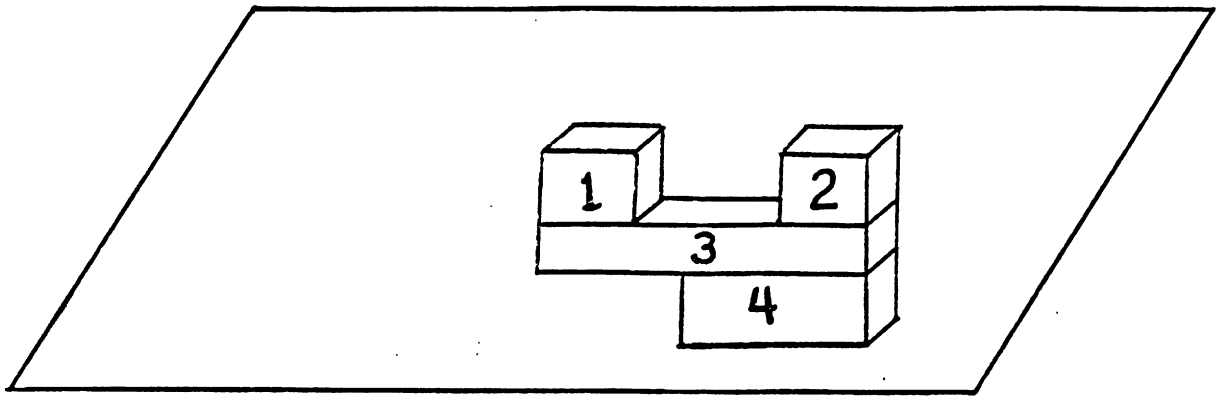
BUILD can deal with plane-faced blocks of any pre-defined shape in any position and rotation in space. The stability-testing routines can deal with any frictional forces that arise. At present, adding a new shape to the system requires that a large number of facts and relations be input by hand, but shape-learning programs of various degrees of complexity are not hard to envision. A very clever program of this type might even notice interesting

relations between the various shapes, such as, "One of these and two of those can fit together to form a brick". For my current work, bricks (perpendicular projections of rectangles) and wedges (perpendicular projections of right triangles) of arbitrary dimensions are sufficient, and are thus the only shapes currently defined in BUILD.

Some examples will perhaps serve to indicate BUILD's level of competence. In each case the problem is to convert the present state SP into the goal state SG. Where SP is not shown, assume that the current state is something uninteresting, like all blocks neatly separated at the side of the table. We will begin with the simple test shown in figure 1-1. First, block A must be moved out of the way. E is then set aside in order to free block B, which is then moved to a position corresponding to block 4 of SG. E is then placed on top of it, corresponding to block 3. Note at this point that block 2 must be placed before block 1, or an instability will result. Note, too, that although both blocks C and D match block 2, C is the better choice because it is free to move and D is not. Therefore, C is placed in position 2 and D is then placed in position 1, completing the plan.

Whenever possible, BUILD tries to leave undisturbed any blocks already in the proper position. In figure 1-2, for

SG-goal state



SP-present state

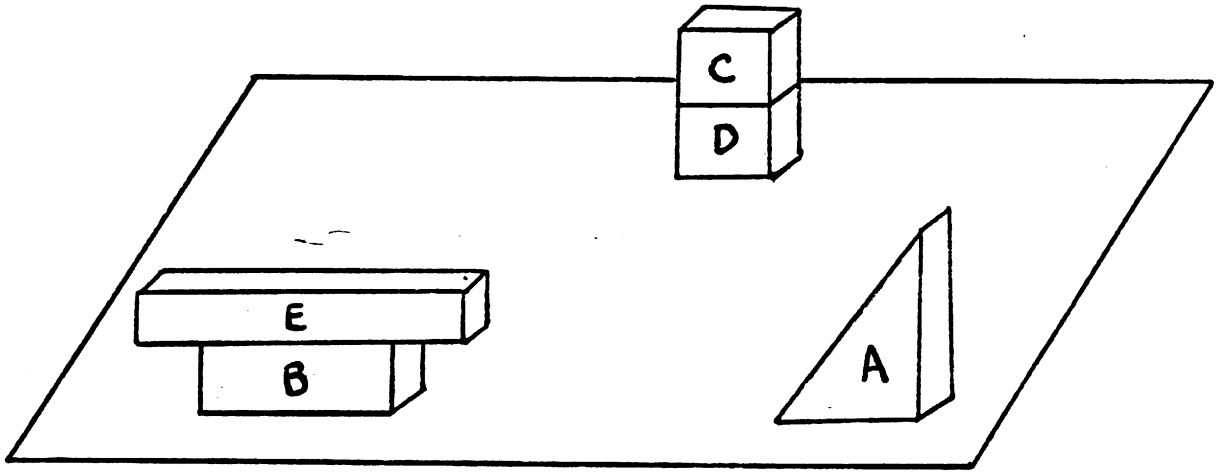
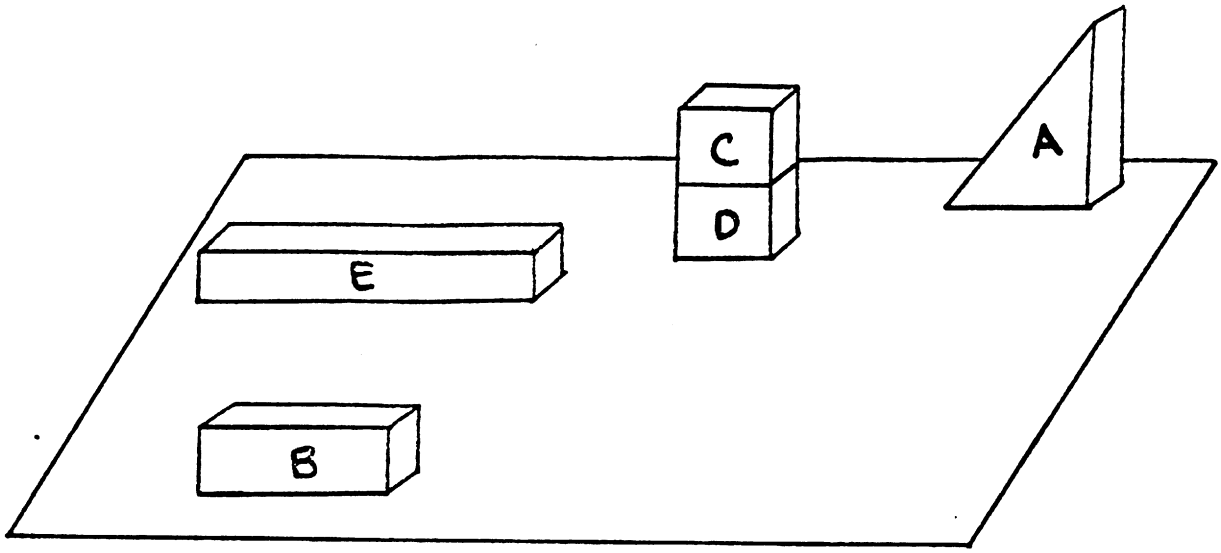


figure I-Ia

basic construction

after step 1 - move E



after step 2- move A

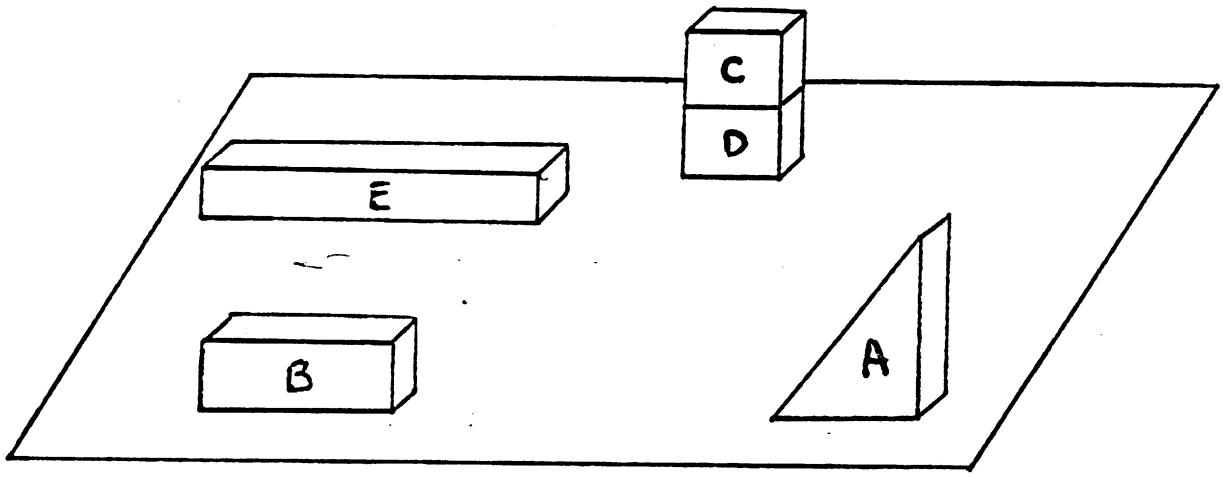
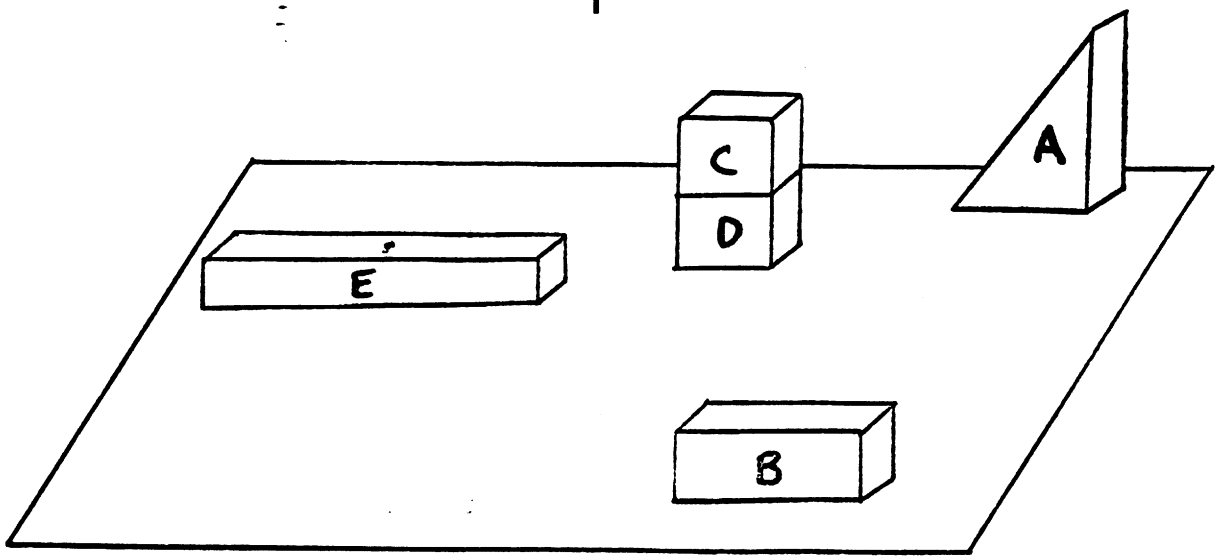


figure 1-1b

after step 3 - move B



after step 4 - move E

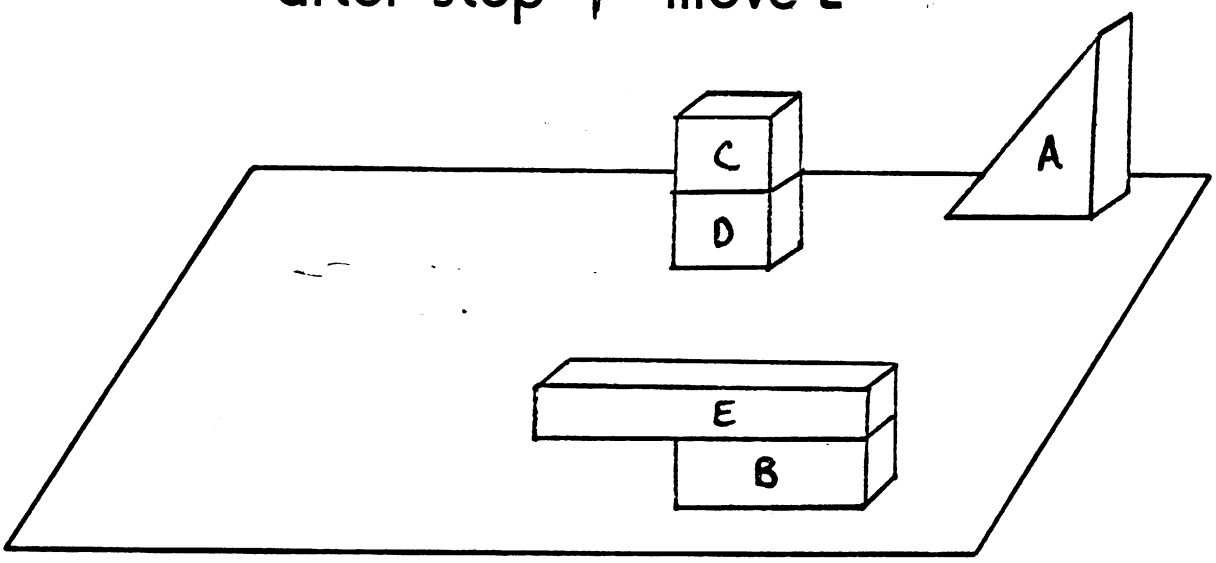
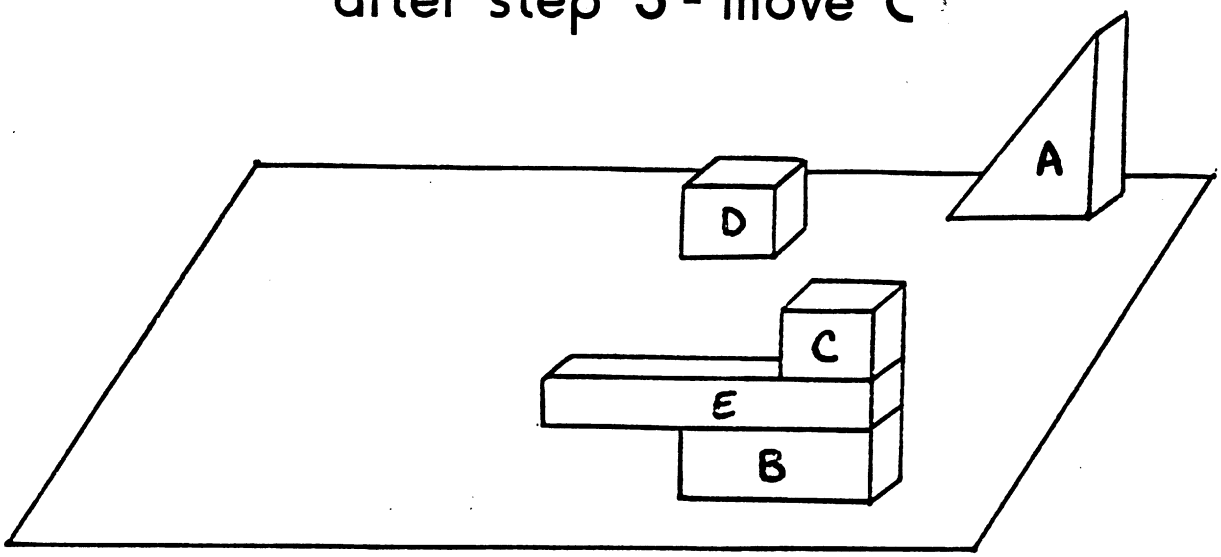


figure I-Ic

after step 5 - move C



after step 6 - move D

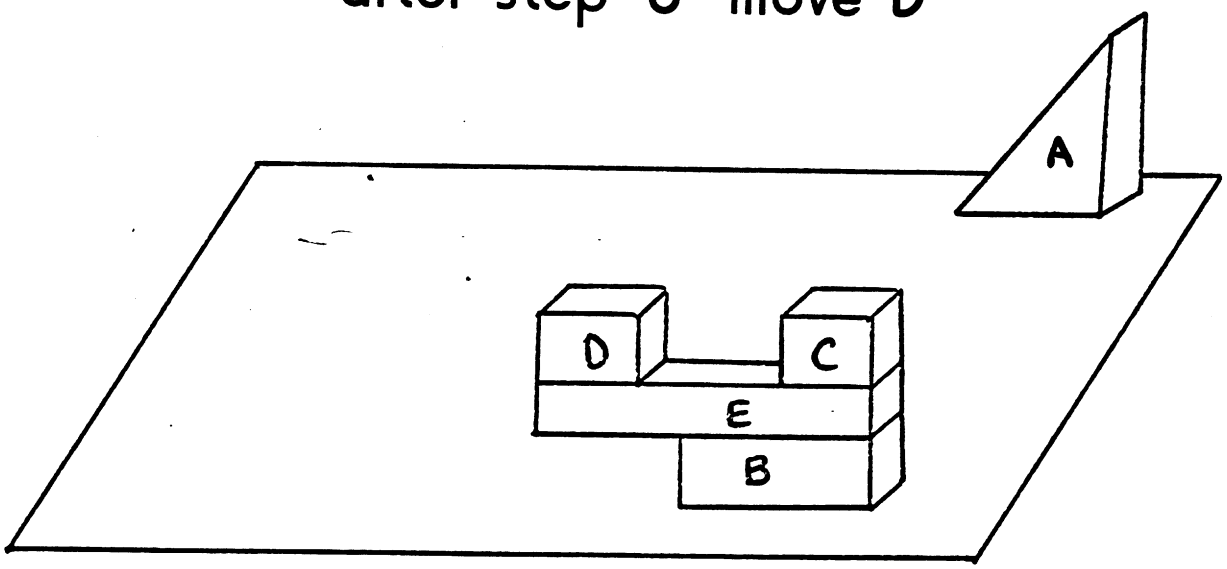
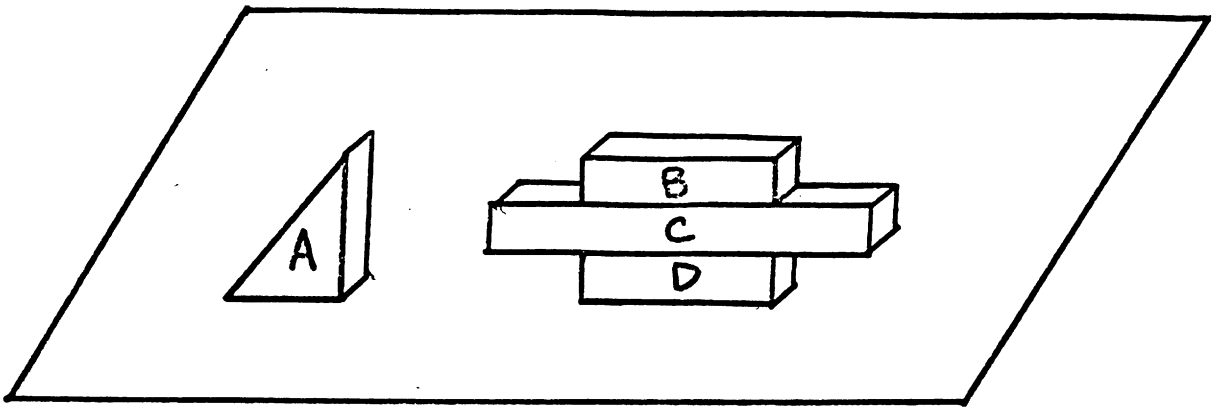
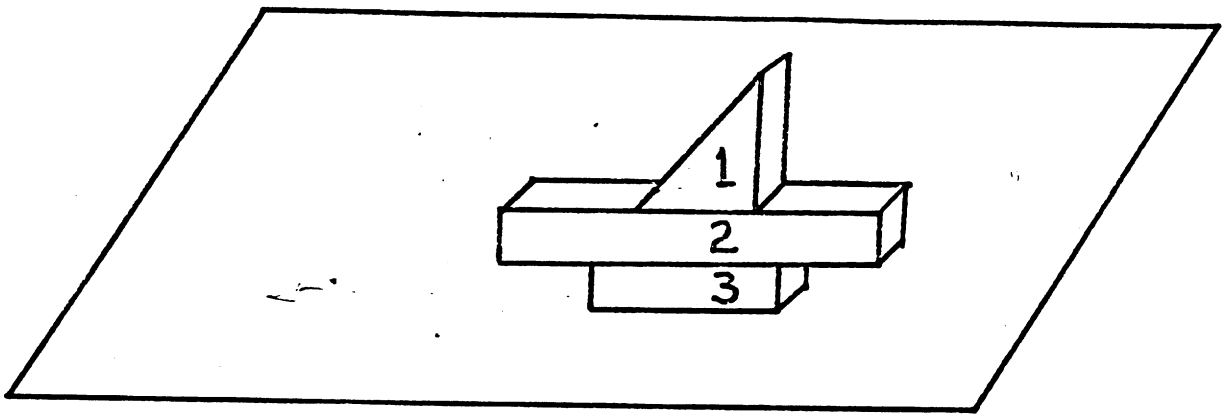


figure I-Id



SP-present state

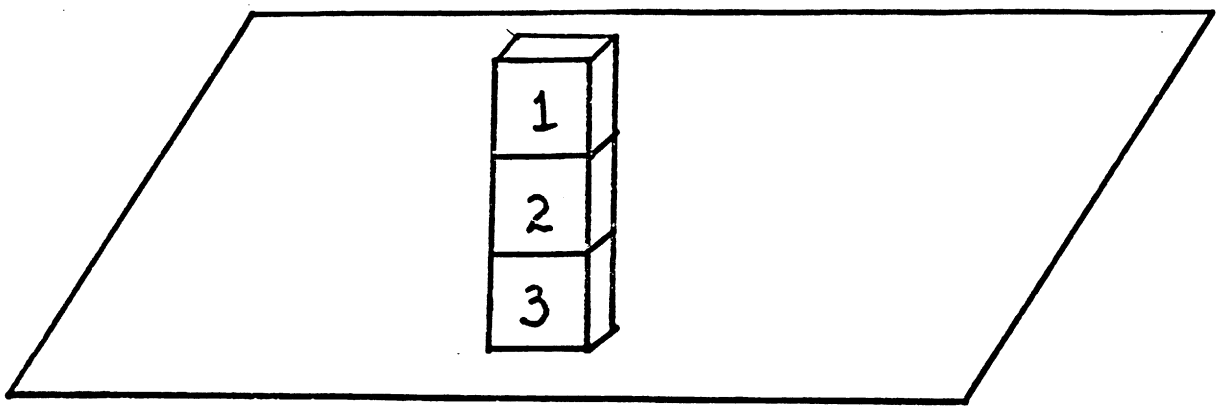


SG-goal state

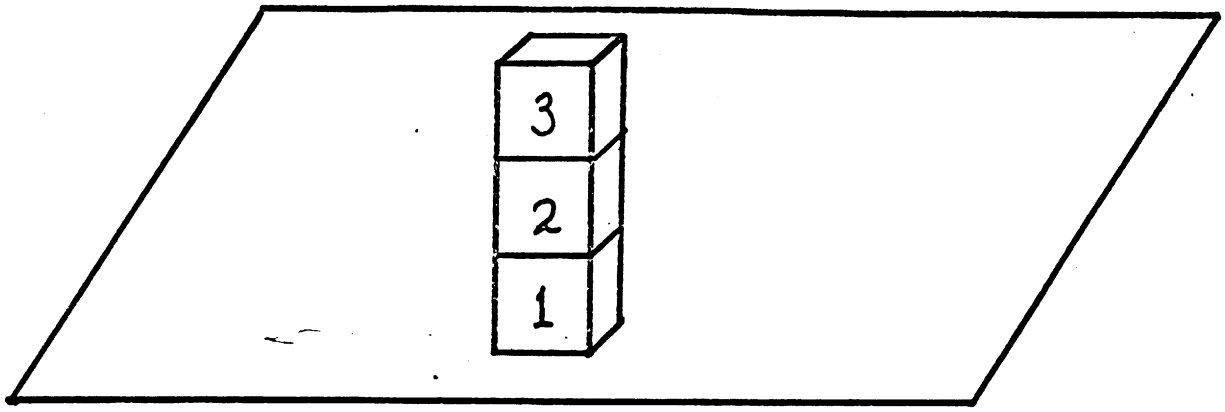
figure 1-2

instance, the plan generated is simply to move B away and to place A in position as block 1. Sometimes, however, a block initially in the proper position cannot be left there, at least without some great extra effort. An example of this is shown in figure 1-3. (If blocks with the same name appear in both SP and SG they will only match each other.) Here, there is no reasonable way to swap 1 and 3 without moving 2. The tower must be completely dismantled and rebuilt in the new order.

BUILD makes use of the concept of movable sub-assemblies of blocks, sets of blocks that can be moved together by the single hand. For a set of blocks to qualify, there must be some block in the set which supports all the others, with no outside supports being necessary. The whole sub-assembly can then be moved by grasping this supporting block. (BUILD does not recognize cases where the hand, by some cleverly chosen grip, is able to grasp more than one block directly.) It is also necessary that no parts of the sub-assembly are so precariously balanced that they will fall off when the structure is moved. To see the usefulness of the sub-assembly concept, consider the problem in figure 1-4. The structure in SG, which I call the seesaw, is one of the classical problems in the area of one-handed construction. Looking first for simple solutions,

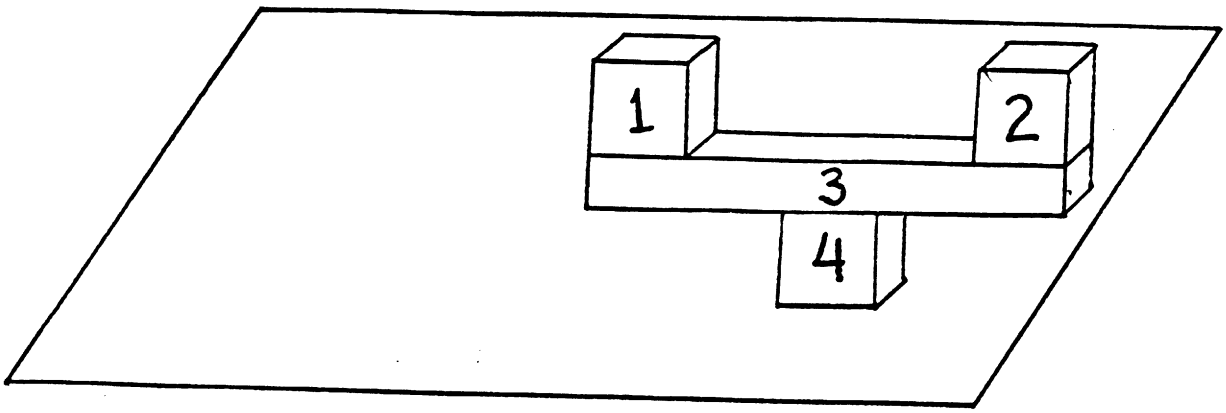


SP-present state

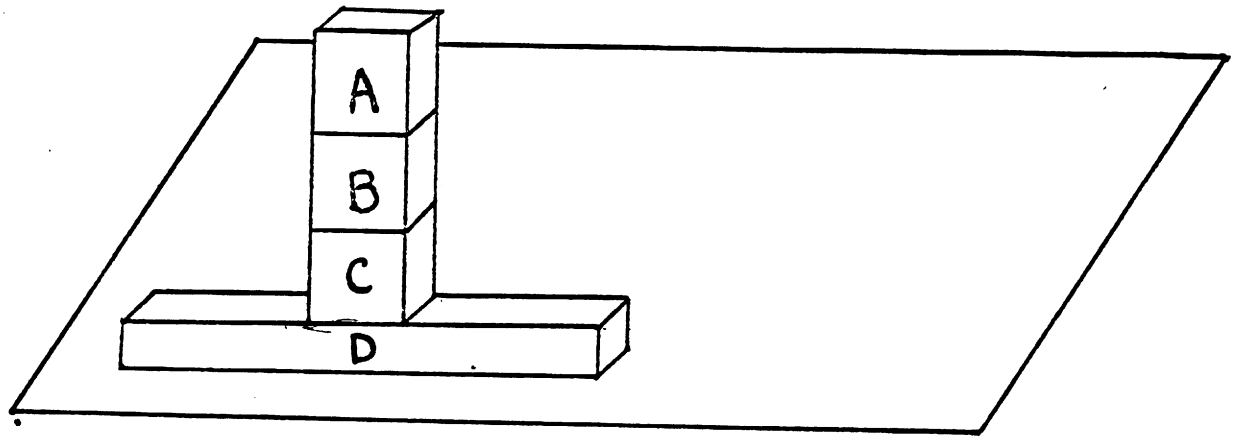


SG-goal state

figure 1-3



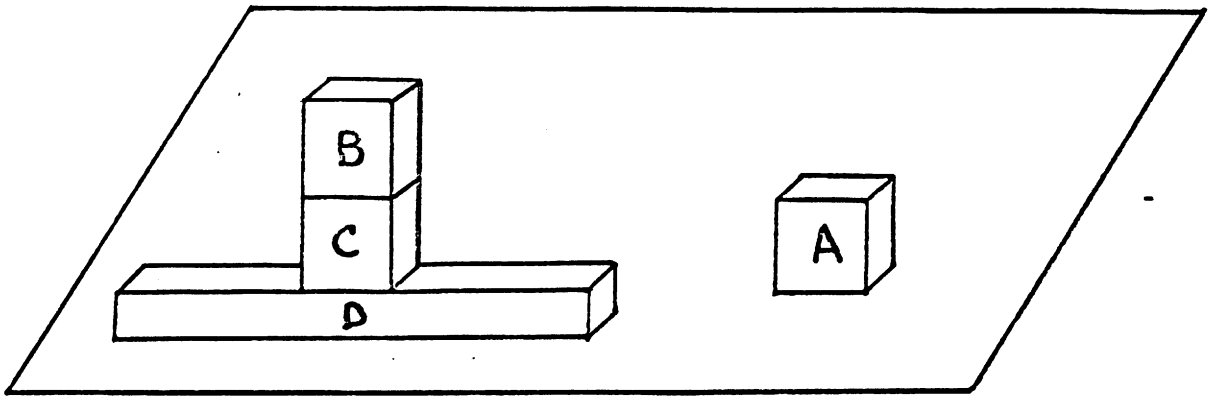
SG-goal state



SP-present state

figure I-4a
sub-assembly

after step 1 - move A



after step 2 - move B

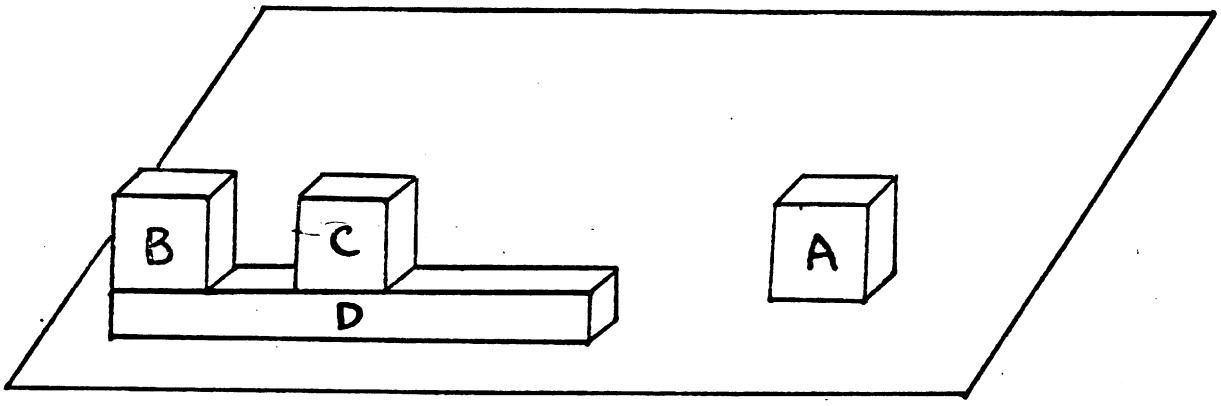
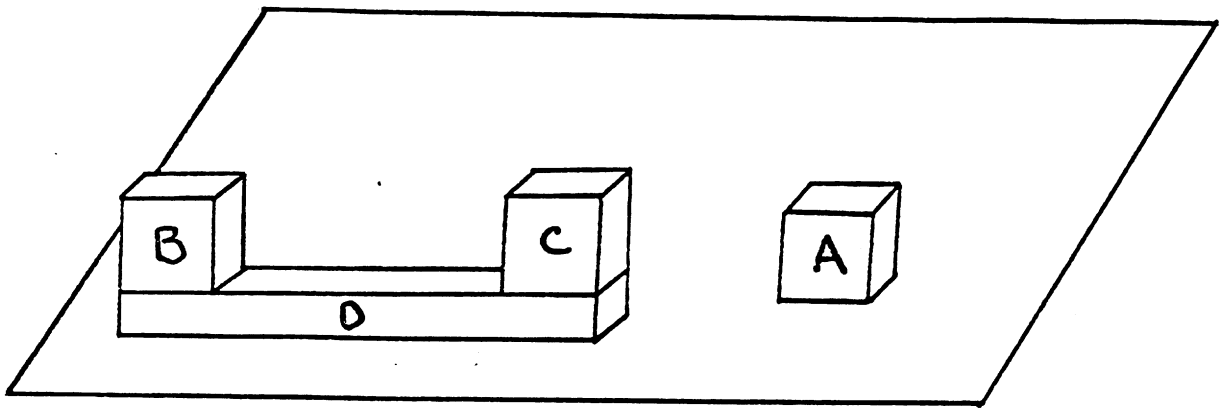


figure 1-4b

after step 3 - move C



after step 4 - move g (BC) riding D

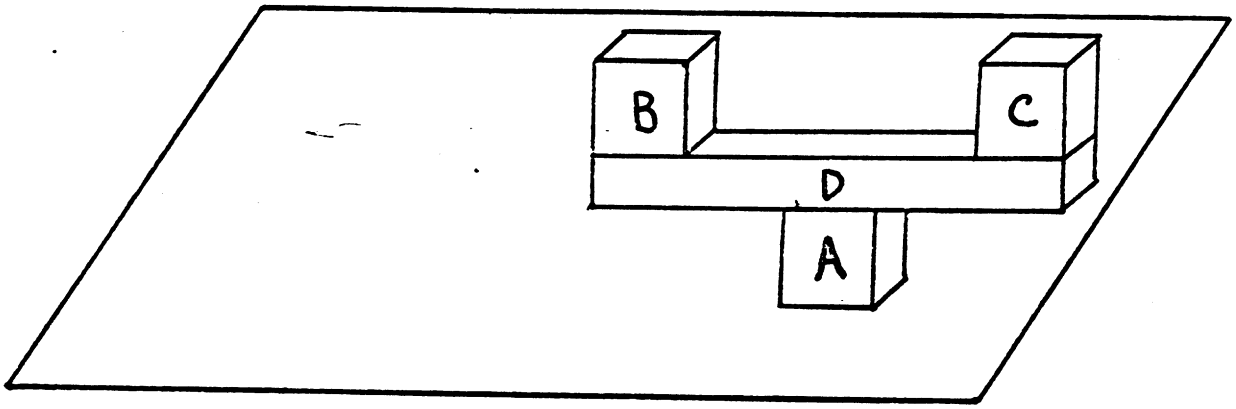


figure 1-4c

BUILD places block A in position 4, sets aside blocks B and C, and places block D in position 3. This, however, leads to a dead end. By further analyzing SG, BUILD discovers that blocks 1, 2, and 3 form a movable sub-assembly which can be assembled on the table, and then, by grasping 3, can be lifted onto block 4. BUILD could, at this point, move block D back onto the table, but since this premature move has occurred only in the model and not in the real physical world, BUILD will instead alter the plan so that the wasted motion is eliminated. To accomplish this, BUILD reconstructs the scene as it existed prior to the decision to dig up and move block D. A spot on the table is chosen in which the sub-assembly will be built, in this case the current position of block D. B and C are placed in the positions relative to D that 1 and 2 hold relative to 3. The structure B-C-D is then grasped by block D and lifted into position. This sequence of actions replaces the sequence "Remove B, Remove C, Place D" in the former plan. If there were subsequent steps in the previous plan, they would now be checked to see if the change has made it necessary to alter other, later steps as well. In this case, there are no subsequent steps in the old plan, and the plan is complete at this point.

Many variations on the sub-assembly idea are possible.

In figure 1-5, the structure A-B-C-D-E is too delicate to be

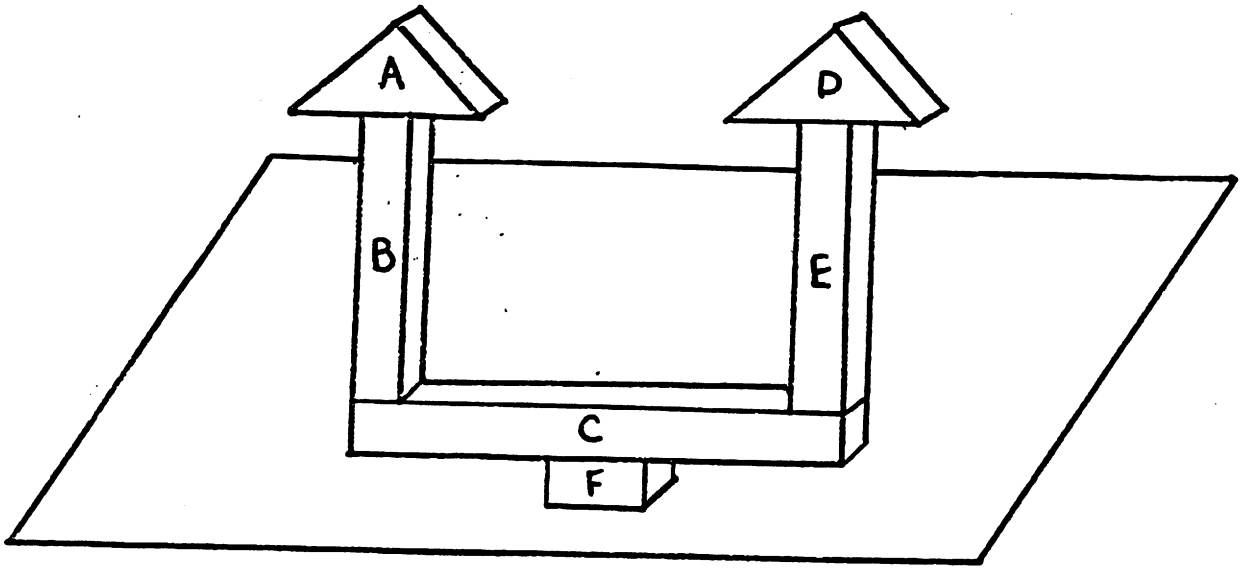


figure 1-5

moved, so the sub-assembly technique is not appropriate. The steadiness of the hand is a parameter supplied by the programmer. If desired, BUILD could be altered to give hand-movement commands that specified how gentle the motion should be.

In figure 1-6, blocks A and B are too precariously balanced to be part of a movable sub-assembly, but can be added after C-E-F has been lifted into place. Note that BUILD might first have worked out a plan to place D, C, B, and A, and only then have discovered that E and F should have been added at the same time as C. BUILD would go back

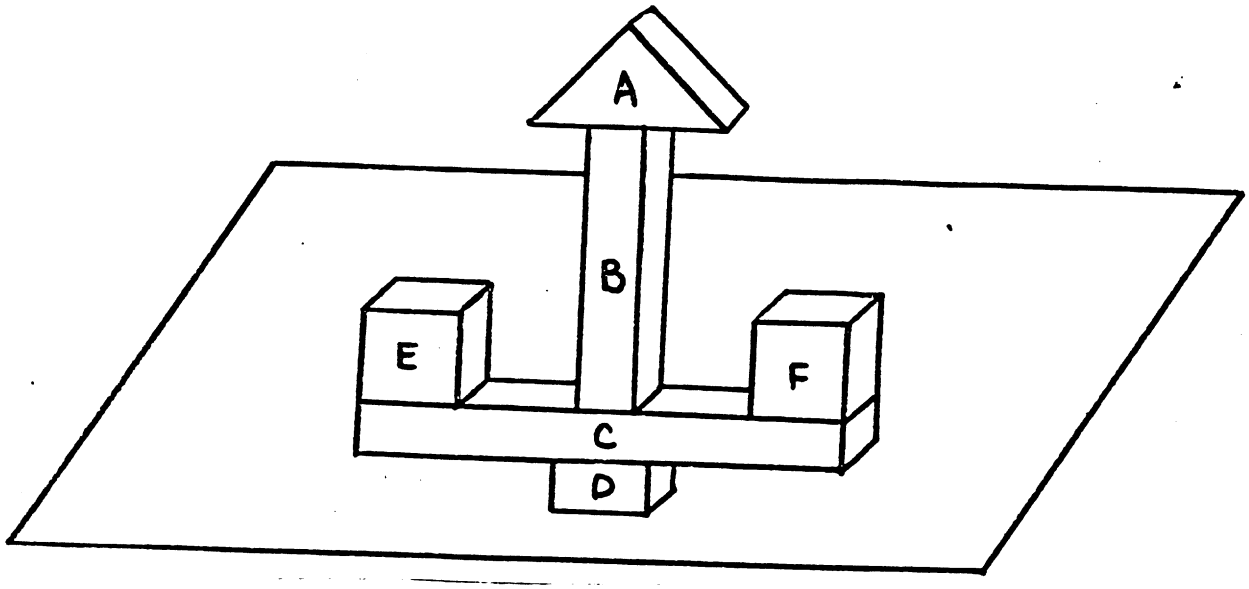


figure 1-6

and modify the "Place C" step and then verify that A and B can be placed as previously planned. If, for instance, a lot of planning went into freeing A and B from their former positions, this verification will be much easier than recomputing the necessary steps.

In figure 1-7, the sub-assembly technique will not work because block F requires support from outside the group, and there is nothing else that will balance block A.

In figure 1-8, the solution is surprisingly easy because BUILD recursively calls itself when constructing the sub-assembly on the table, and thus has available all of the power that it can use on its top-level goals. G-H-I is

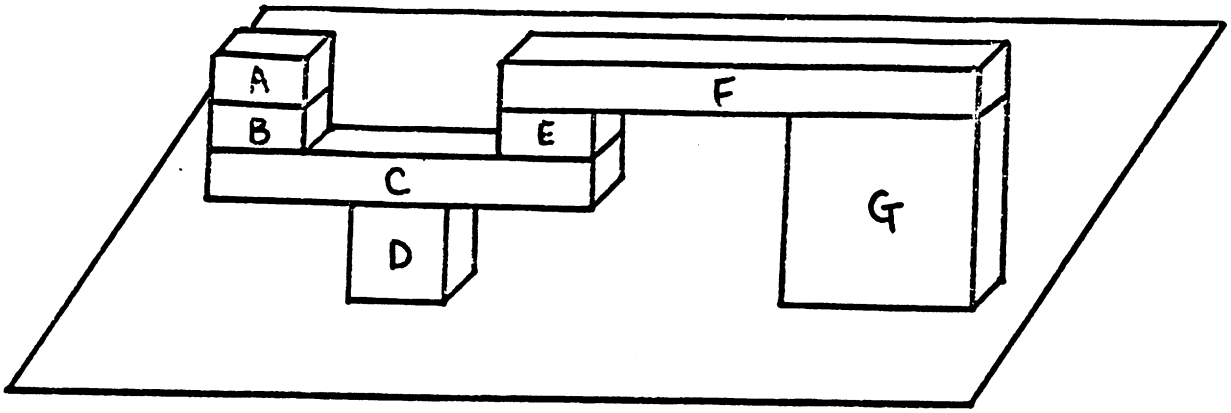


figure I-7

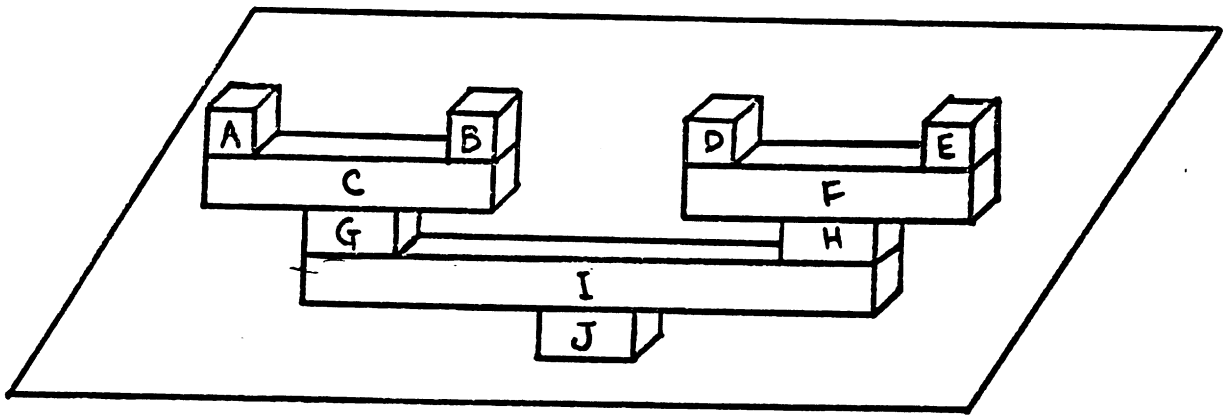


figure I-8

built on the table. The two mini-seesaws are built on the table and lifted into place. Finally, the entire sub-assembly is lifted and placed onto J.

Figure 1-9 shows a problem for which the sub-assembly

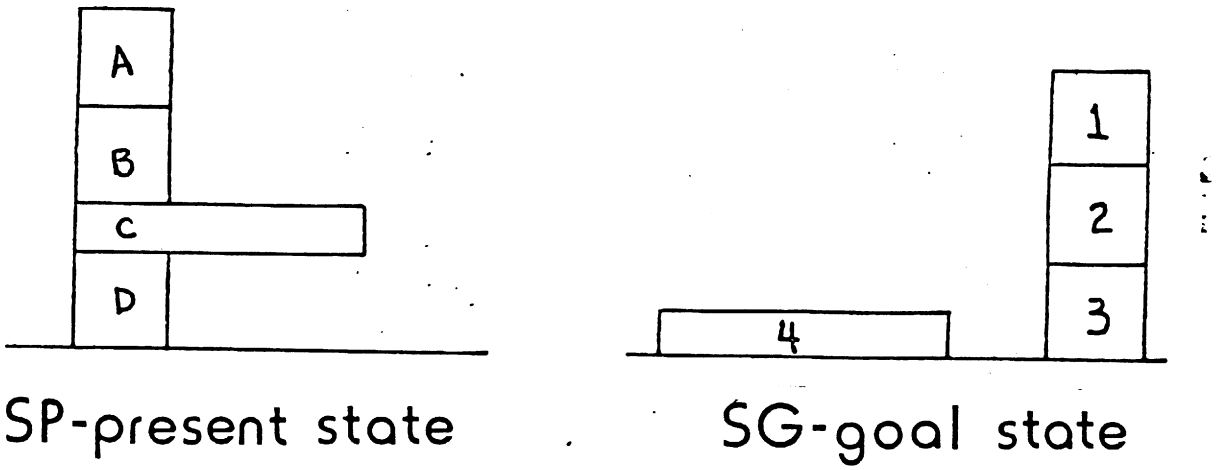


figure 1-9

technique must be used to dismantle a structure rather than build one. Because it can neither look again at the scene nor predict where the blocks will fall, BUILD will not simply knock down difficult structures. Instead it reverses SP and SG, calls itself recursively to see how the offending structure could have been built, and reverses the ensuing plan. In this way, all of the construction techniques can be used for destruction as well, without the necessity of

keeping around both forward and backward versions. In this example, A-B-C is lifted down to the table, and then dismantled.

Another of BUILD's methods for dealing with difficult structures is the use of extra blocks in the scene as temporary supports and counterweights. This method may often be used in cases such as those in figure 1-10, where sub-assembly is not possible. Figure 1-11 shows the use of a temporary support in the construction of a seesaw. Figure 1-12 shows a larger block being used as a counterweight in the same construction.

The temporary structures used can take many different forms, depending on the shape and dimensions of the extra blocks available at the time. Several of these forms are shown in figure 1-13, with the temporary blocks marked by an X. A and B are simple scaffolds made of several blocks. In C the wedge is used to provide a variable extra height for the tower. Of course, the wedge must not slope too steeply or it will be pushed off to the side. In part D the unstable parts are blocked from above rather than below. E shows a multi-block counterweight, while F shows a tower resting on one of the permanent blocks rather than the table.

The place chosen by BUILD for construction of a

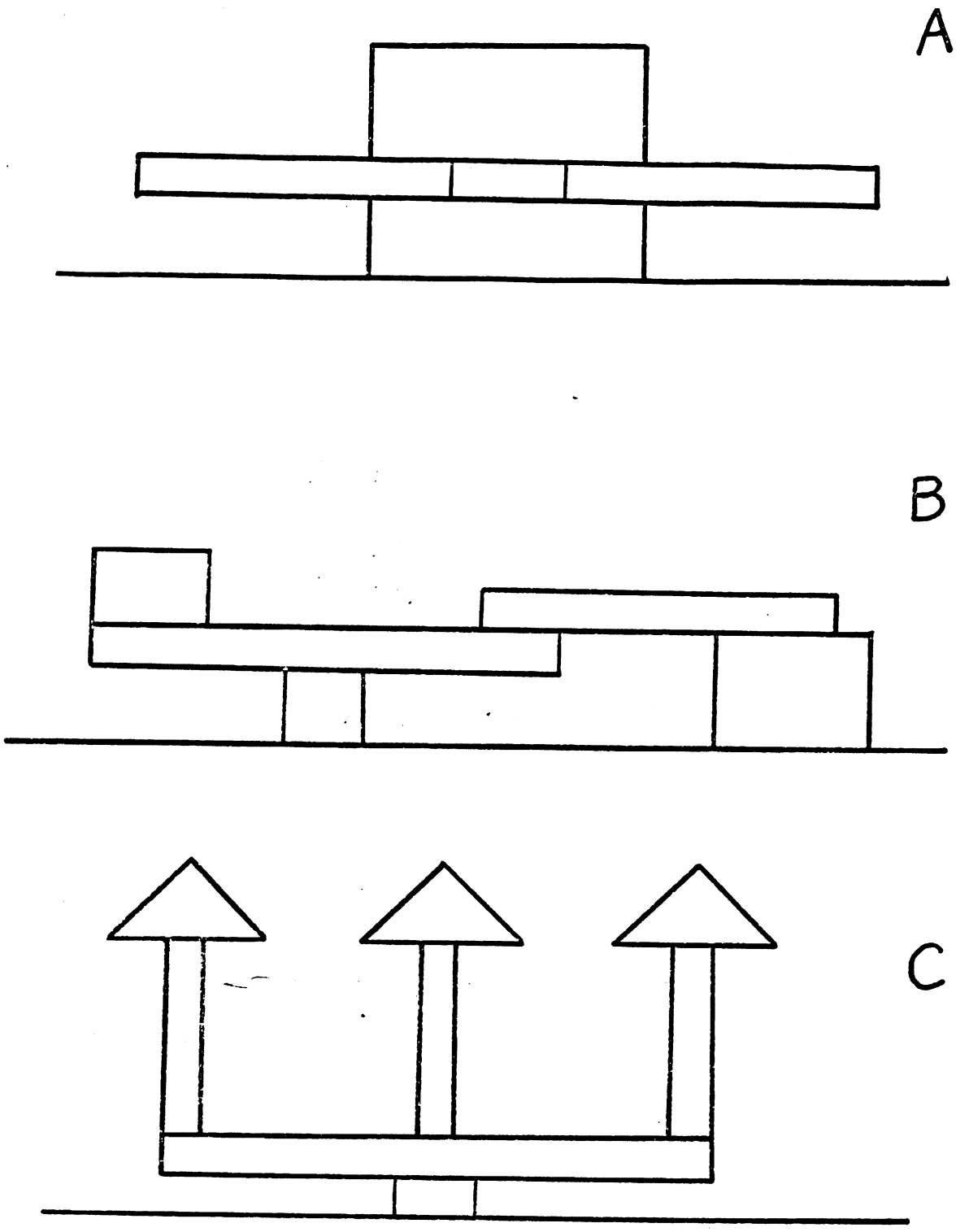
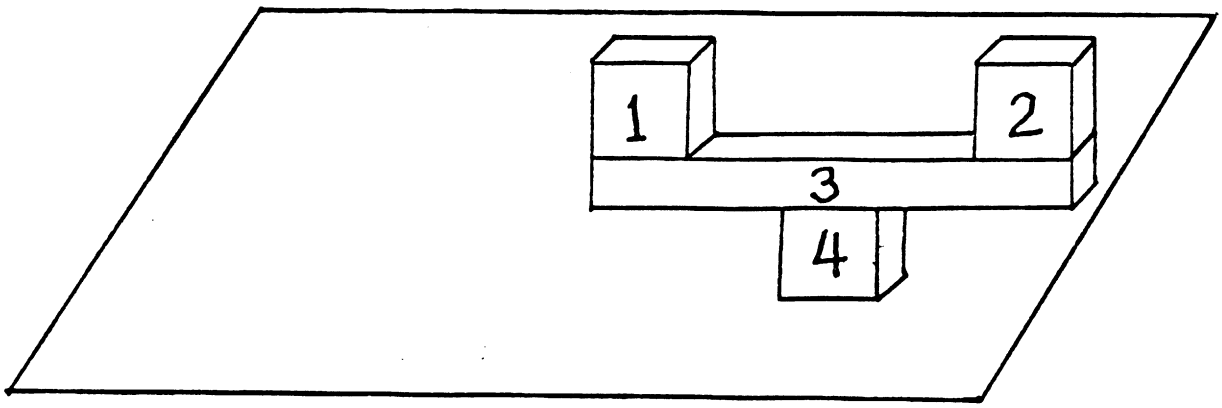


figure 1-10

SG-goal state



SP-present state

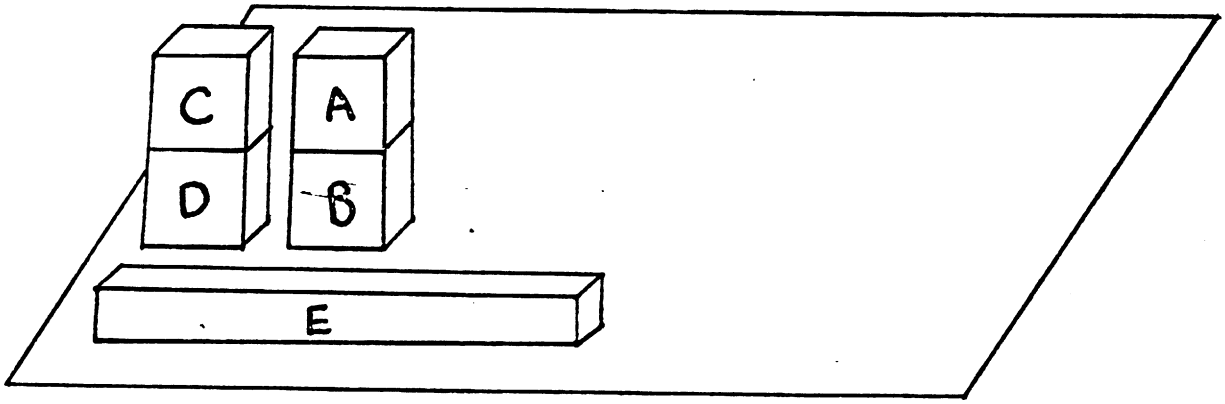
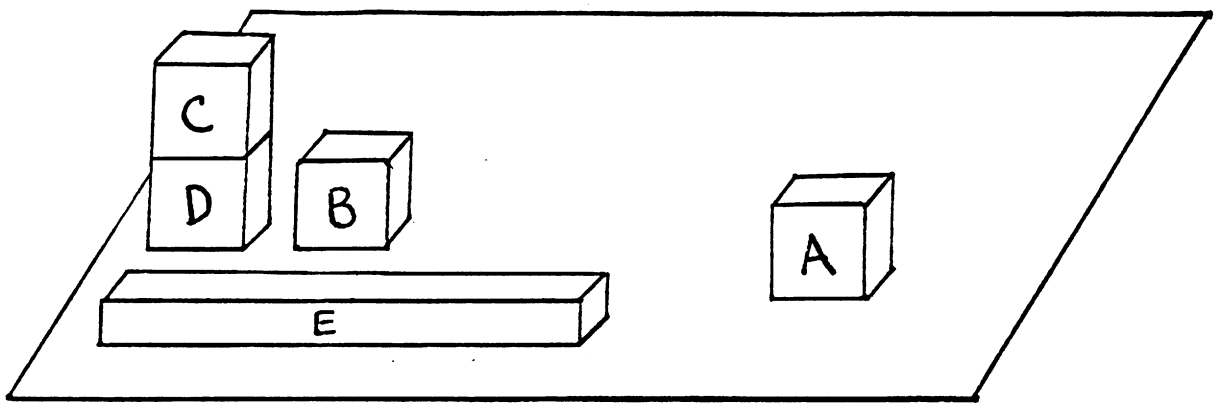


figure I-IIa

scaffold

after step 1 - move A



after step 2 - move E

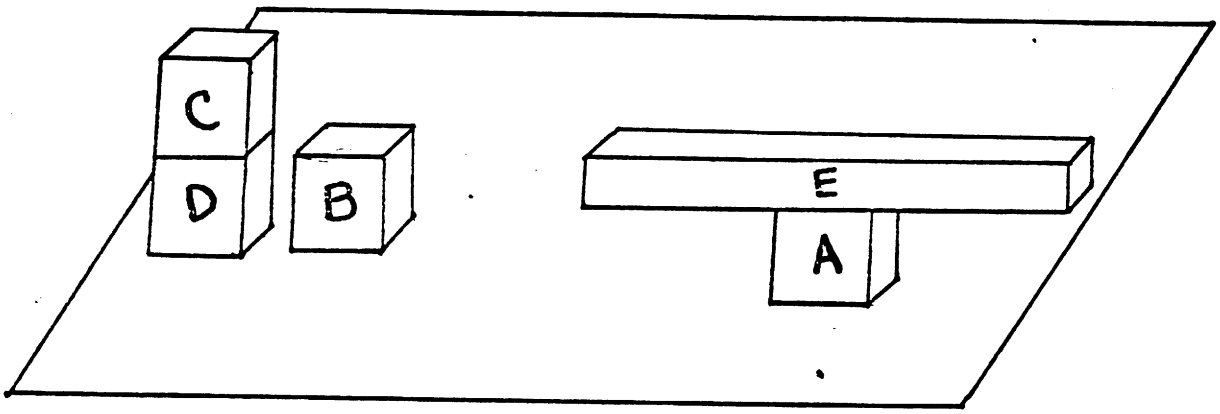
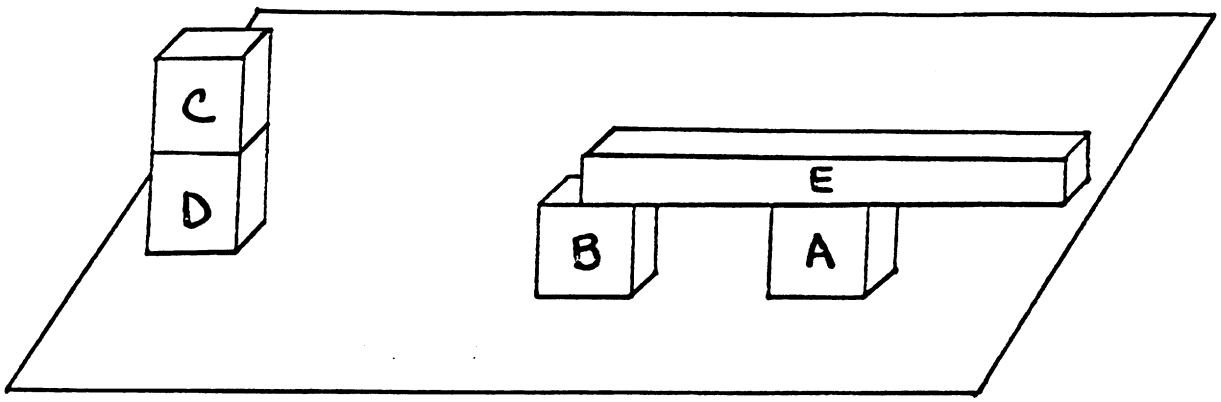


figure I-11b

after step 3 - move B



after step 4 - move C

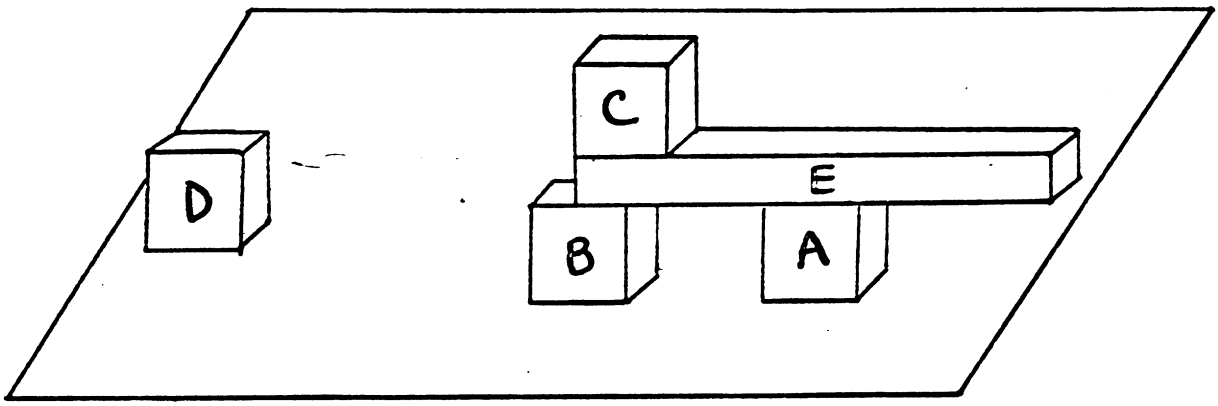
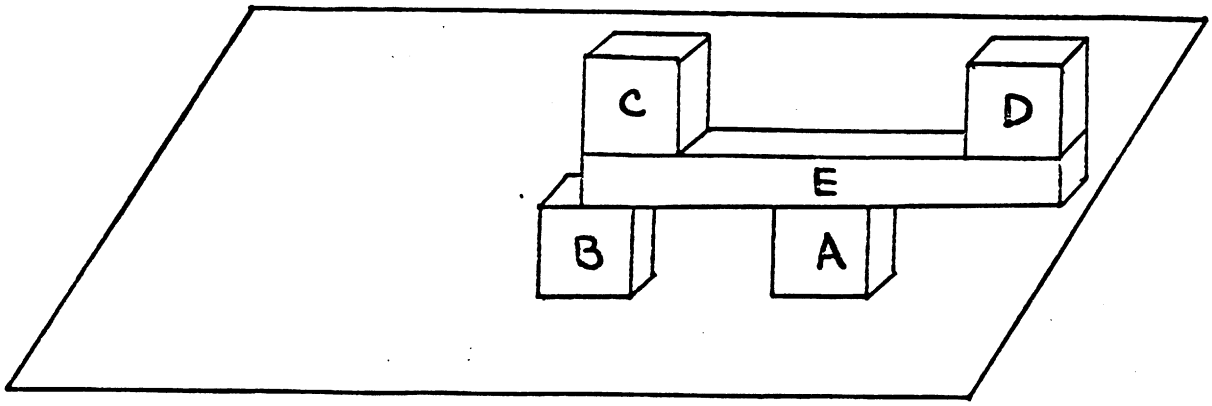


figure I-IIc

after step 5 - move D



after step 6 - move B

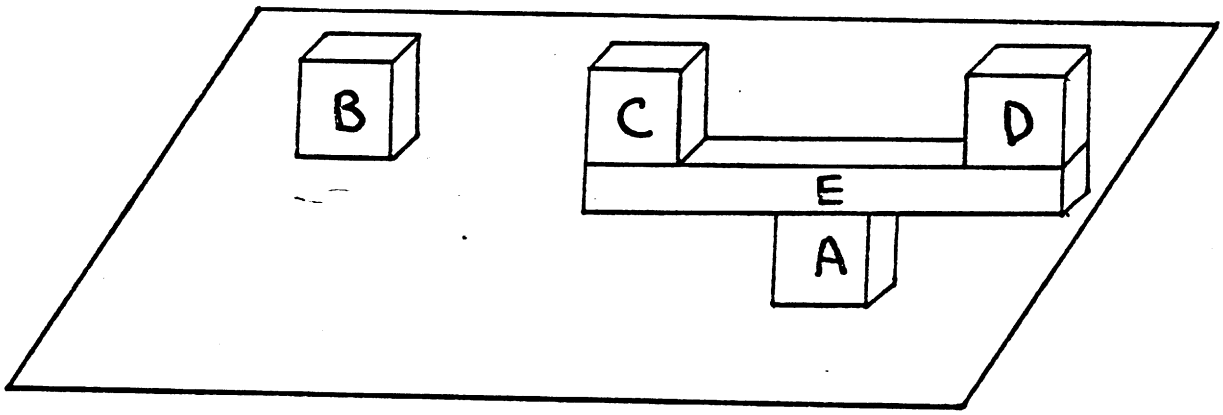
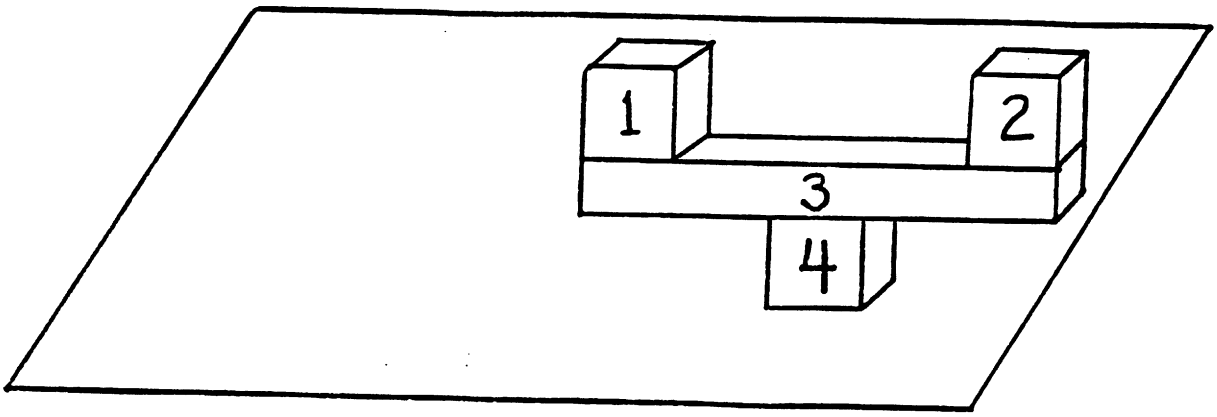


figure I-IId

SG-goal state



SP-present state

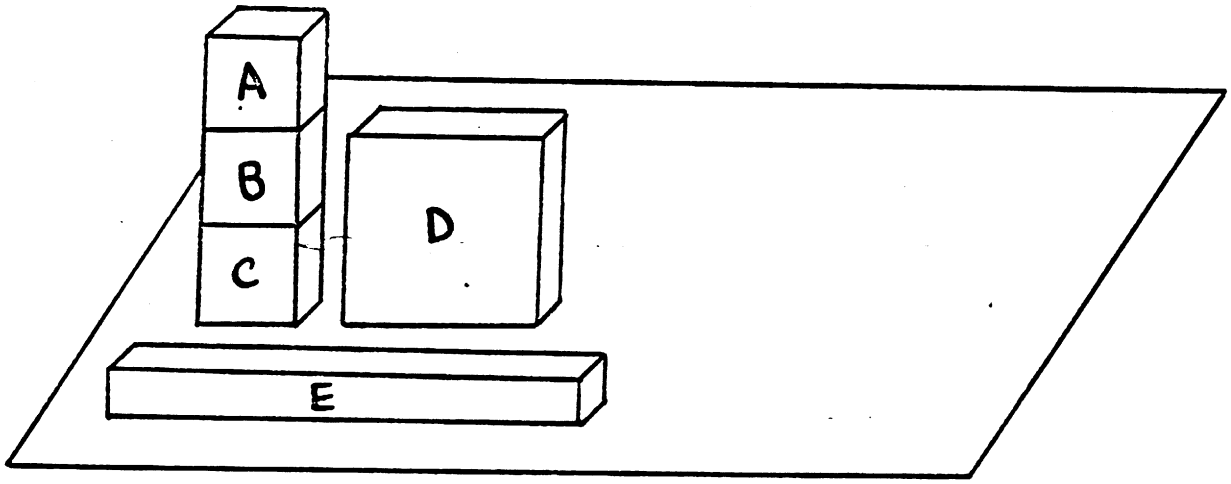
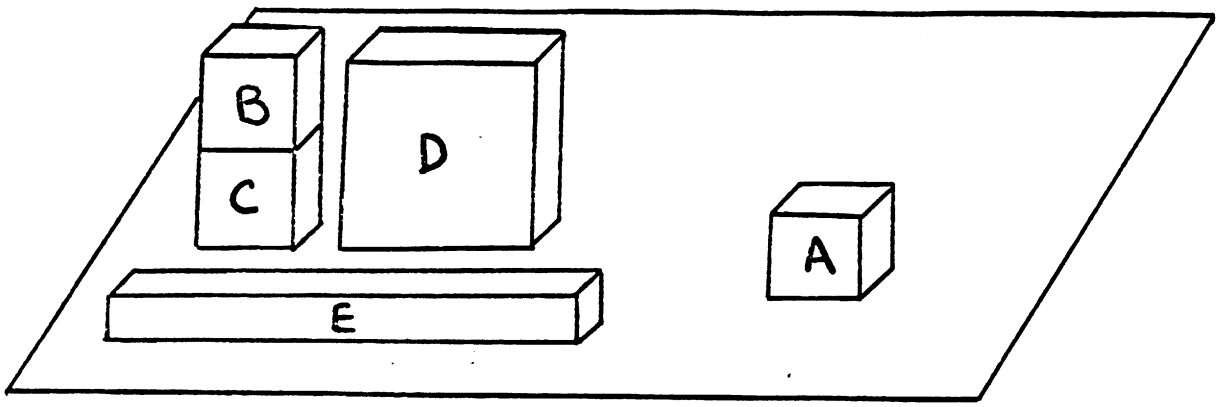


figure 1-12a
counterweight

after step 1 - move A



after step 2 - move E

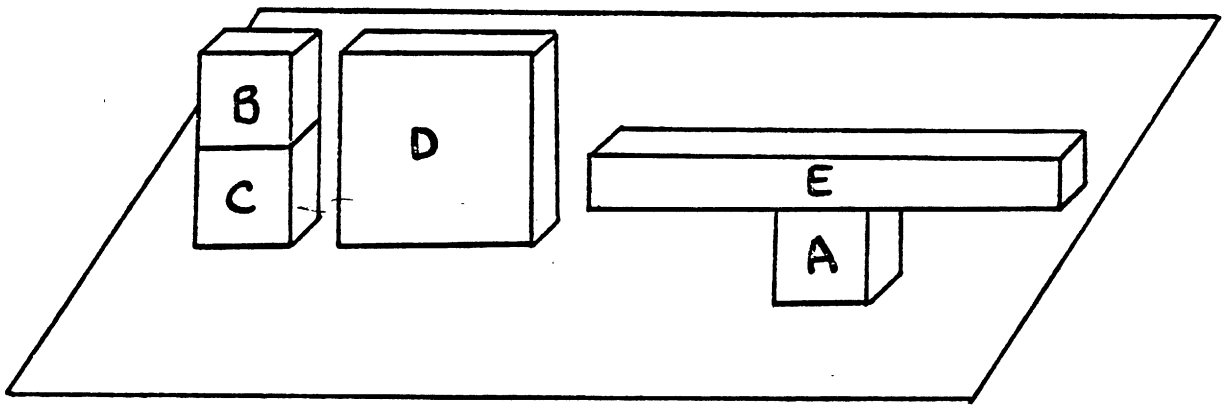
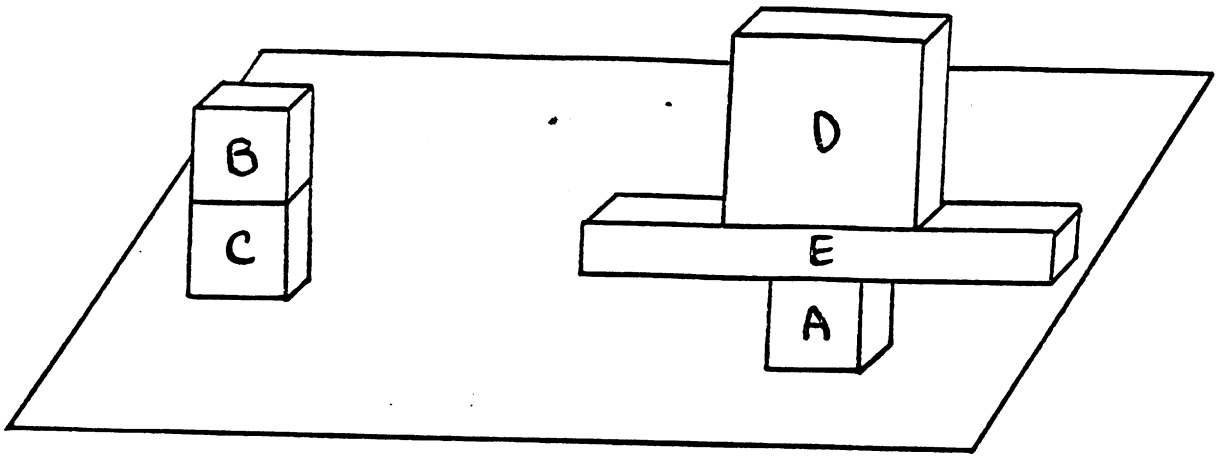


figure 1-12b

after step 3 - move D



after step 4 - move B

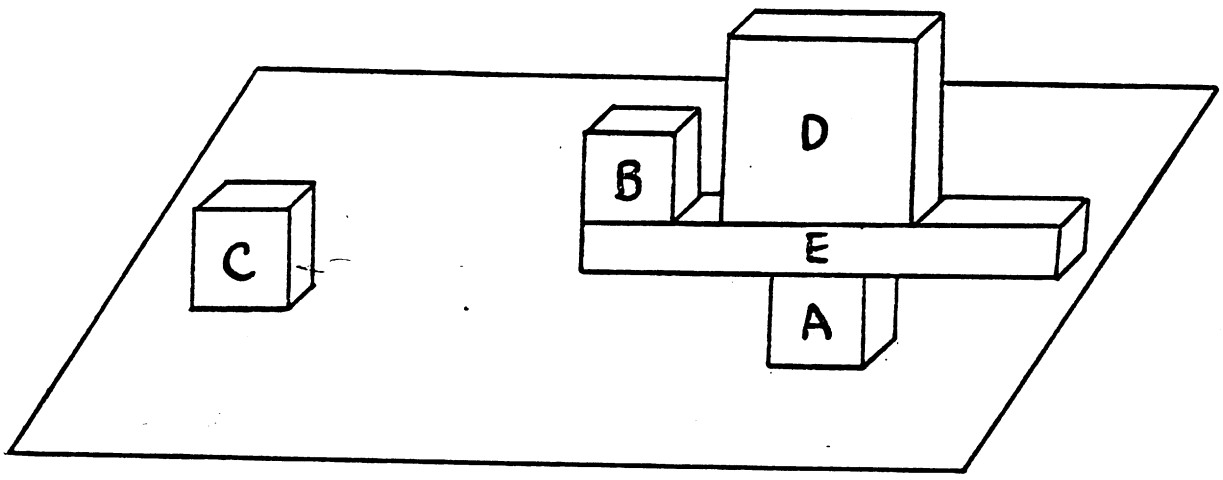
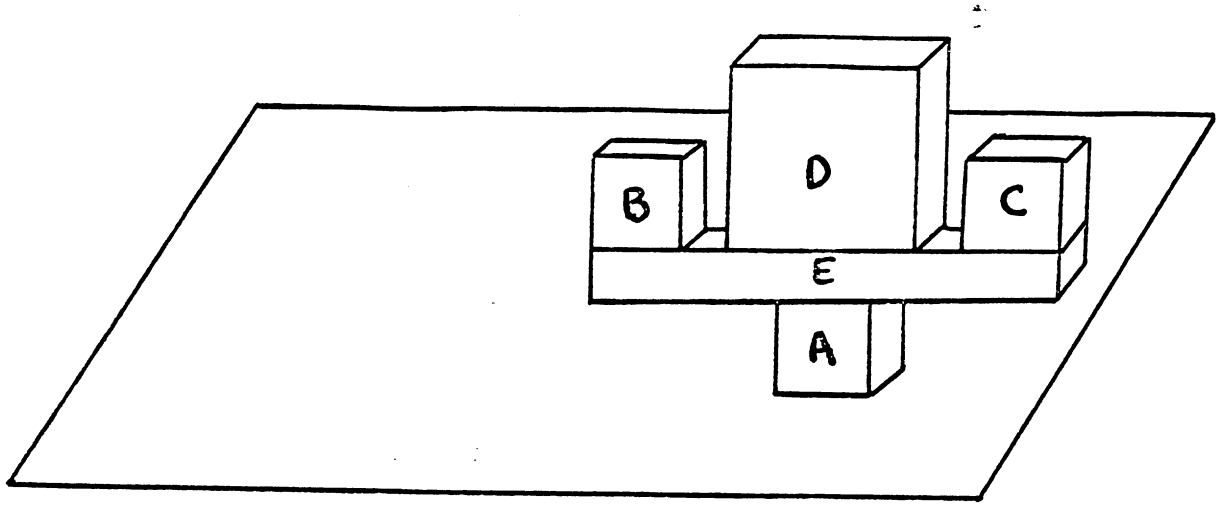


figure 1-12c

after step 5 - move



after step 6 - move D

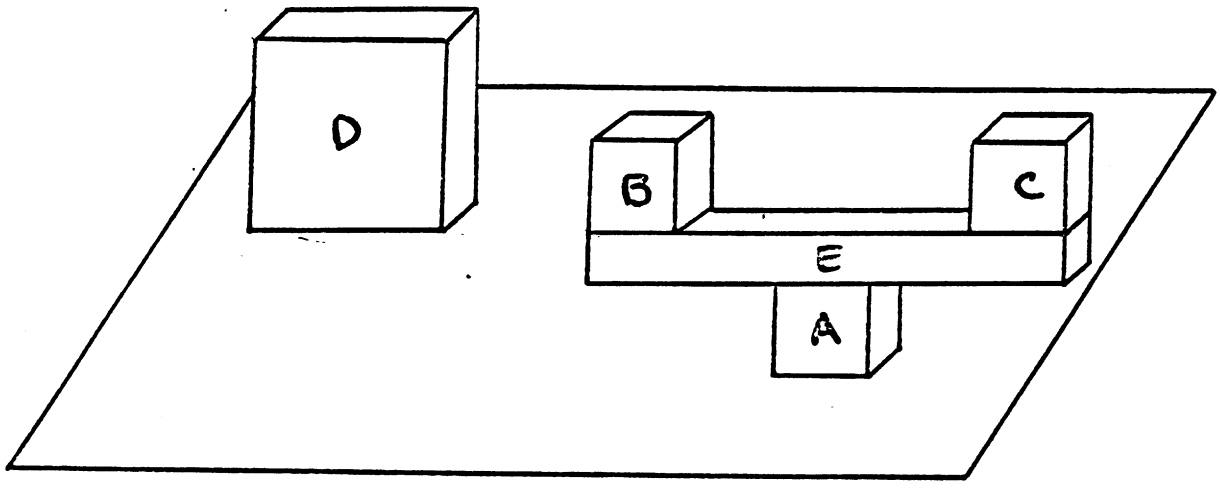


figure I-12d

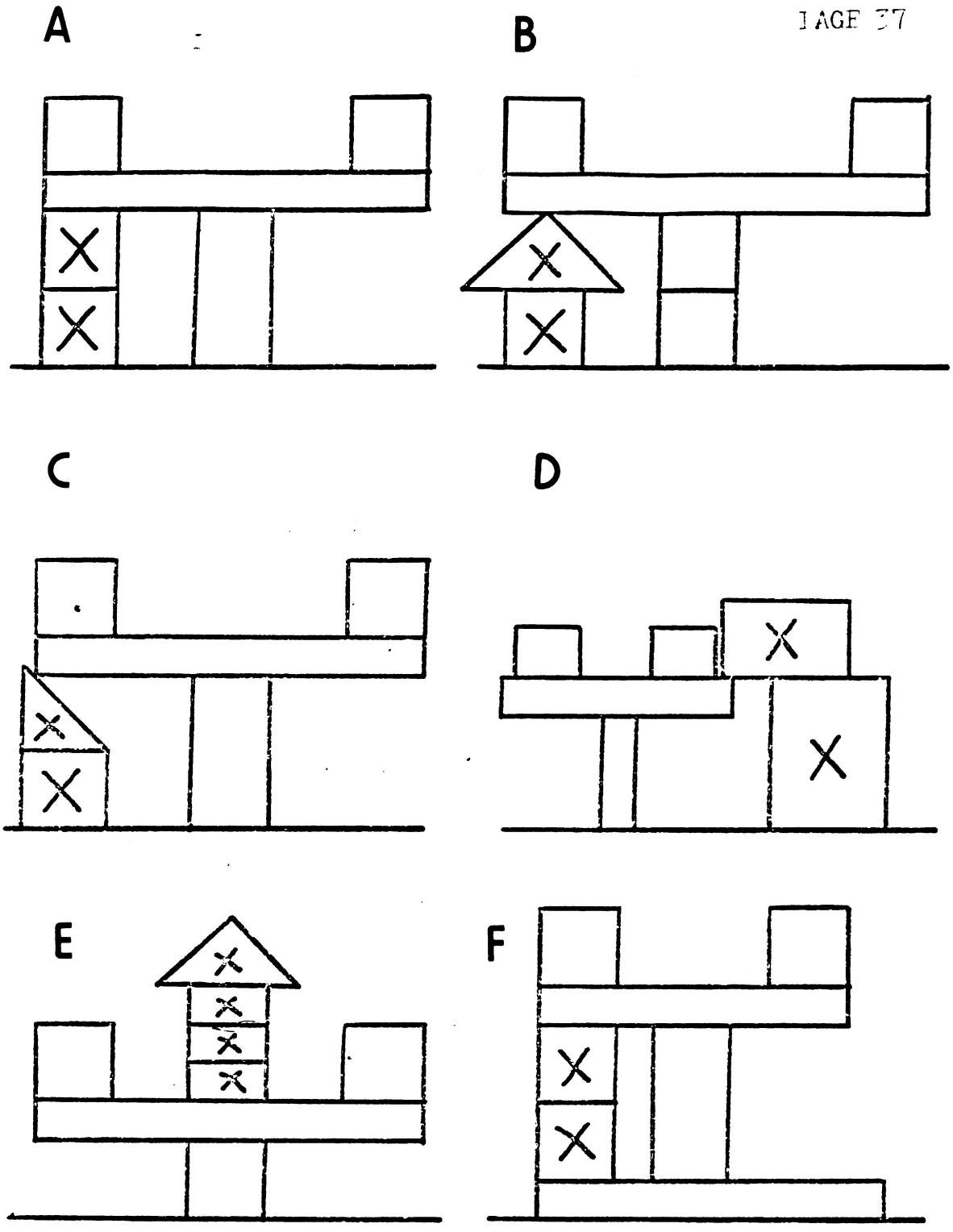
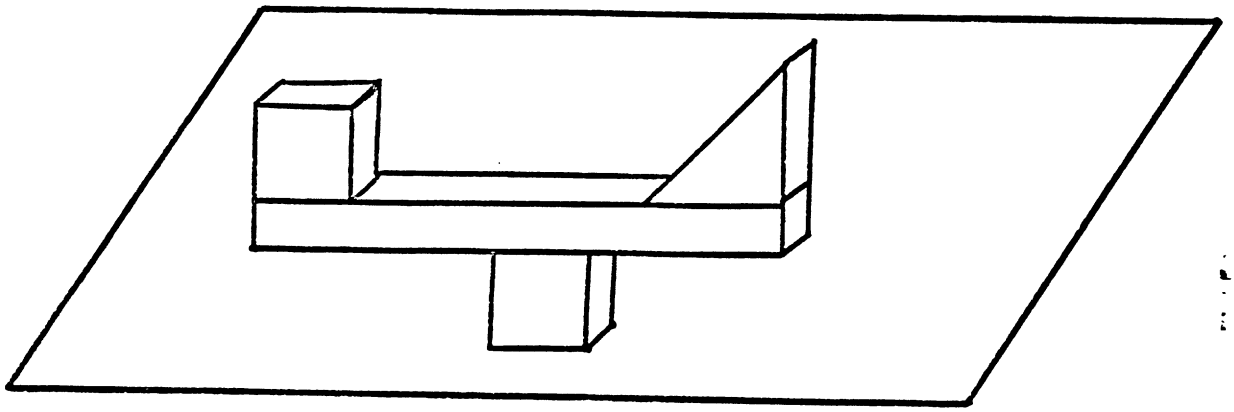


figure I-13

temporary structure may already be occupied by other blocks. In this event, BUILD must choose one of several options: It may be possible to build the structure elsewhere, either nearby or, for instance, under the opposite side of a seesaw. If the offending blocks are steady enough and present a fairly level upper surface, the structure could be built on top of them. It is, of course, usually possible to remove the intruding blocks, or to alter the sequence of the plan so that the temporary structure is built, used, and dismantled before the other blocks are ever placed. The raw materials for the temporary structures can, if absolutely necessary, be taken from other existing structures. Once again, plan alteration can be used to minimize wasted movements due to premature placement of certain blocks. Care must be taken not to fall into loops where two structures each require the temporary use of parts from the other.

One fairly obvious method that BUILD does not employ is the technique of sliding a block or changing its position while maintaining its pressure on some other block. Figure 1-14 will demonstrate what I mean. If the wedge were picked up completely, the seesaw would topple. An obvious solution is to rotate the wedge in place, maintaining its pressure on the cross-arm. BUILD is unable to do this, and instead

SG-goal state



SP-present state

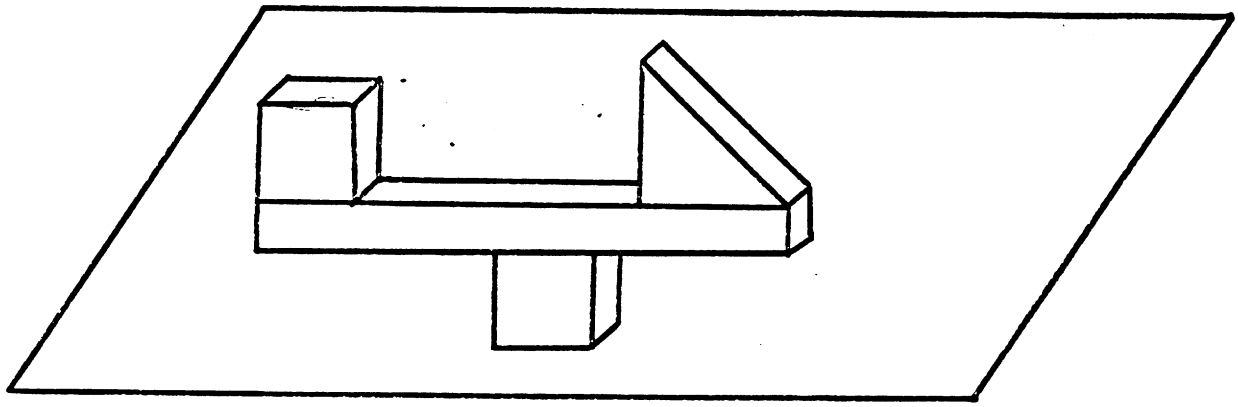


figure 1-14

would lift the sub-assembly to the table, remove the wedge, replace it in reversed position, and then replace the whole sub-assembly. The problem with implementing sliding moves is simply that there is no general way of determining whether an uninterrupted sliding path exists without attacking the path-finding problem as a whole and, as I said earlier, this is an issue I have chosen to avoid.

Because of the visual simplicity of blocks, they are the favorite subject matter of computer vision researchers at M.I.T. and elsewhere. The M.I.T. vision group has demonstrated the capability of looking at very simple block scenes with the computer's vidisector eye and copying them with a mechanical computer-driven hand. The construction planning portion of this system consists of some simple heuristics involving above-below relationships between block images on the 2-dimensional retina. The system does not make use of any real 3-dimensional models of the scene before it. A secondary goal of my work, then, is to provide BUILD as a much more powerful planning module for this system. Some very exciting possibilities exist for cooperation between the vision-hand system and BUILD: Visual feedback can tell BUILD whether its plans are having the desired effect, as predicted by BUILD's internal model. Meanwhile, BUILD's ability to analyze the support and touch

relationships in a model can help the vision system choose the most reasonable of several possible interpretations of a scene. Since Terry Winograd's natural language understanding system <WINOGRAD 1972> uses 3-dimensional models of blocks and block structures as subject matter, BUILD could be fairly easily interfaced to these as well. This would result in a computer system that could receive commands in plain English, do some fairly complex planning, and execute these commands in the real world.

While BUILD does not do anything that could be properly considered learning, I feel that it provides a good laboratory for research in this direction. Plans, as well as structures and individual blocks, could be compared and classified by a description-comparing program similar to the one described by Winston <WINSTON 1970>. By this means, plans that are demonstrated to the programs or derived by inefficient general methods can be generalized and saved as special-purpose techniques to be used again when similar situations arise. BUILD can provide the necessary framework and utilities to make such work possible.

Most of the planning for BUILD was done with the intention of implementing the system in MICRO-PLANNER. There were, however, some fairly serious problems associated with the use of this language in my particular application.

CONNIVER, which appeared while this investigation was in progress, seemed to solve many of these problems for me, so BUILD was converted to this new language. I will have some comments in section 5.1 concerning the relative merits of the two languages for this type of problem-solving.

To some extent, BUILD can be considered a descendant of the "blocks world" portion of Terry Winograd's language system. Both systems, after all, solve problems about blocks using 3-dimensional models. However, since Winograd's main interest was in language rather than in construction problems, his models were so restricted that the type of problems discussed in this section could not even be represented, let alone solved. Therefore, Winograd's actual programs have been useful to me only as inspiration and as examples of good problem-solving style. The Stanford University hand-eye project has made some use of 3-dimensional models <PAUL, FALK, FELDMAN 1969>, but their publications to date do not indicate that they have used these extensively in construction planning, except in the area of collision avoidance.

Chapter 2: The Modeling System

2.1 Overview

BUILD can be roughly divided into two major parts. One part, the planning system, consists of the network of CONNIVER programs that control all of BUILD's planning and decision making. The other part, the modeling system, contains the programs that do all of the work: creating and updating the models, testing blocks for collision and contact, testing structures for stability, finding empty places to put things, and so on. The modeling system will be described first, since the planning system can be better explained once it is clear what information it has to work with and what resources it can call upon. This chapter will cover various aspects of the models themselves; chapter 3 will cover the touch and collision test; chapter 4 will describe the stability test.

Though the physics and geometry of the blocks world are simple and well-understood, the programming of the modeling system was not particularly straightforward. BUILD's entire understanding of its world resides, in the form of procedures, in the modules of this system. The programs must be accurate and comprehensive enough to deal correctly

with all situations of interest, and must provide the planning system with a rich enough flow of information to make intelligent decision-making possible. If, for example, the stability test were a simple win/lose predicate, BUILD would be forced to use a strategy of blind trial and error. Because of these requirements, some of the simpler algorithmic methods for touch and stability testing had to be abandoned in favor of more complicated methods with a higher information yield. The various methods considered for each test are described below, along with a description of the methods ultimately chosen.

2.2 Information Management

The information used by BUILD can be divided into two types according to its source. One type, which I will call primary data, is received from outside and is irreplaceable. Such things as the size, shape, and initial position of each block in a scene are primary data, at least until BUILD is connected to a vision system which can obtain such data from the environment at will. The other class, secondary data, is much larger, and consists of data items that BUILD can derive from the primary data by various routines resident in the modeling system. Some examples of data items in this

category are the three-dimensional positions of the block vertices and face planes, information about the points of contact between blocks, and the network of support relations found by the stability test.

Since some of these secondary data items are relatively costly to derive, it would be wasteful not to save the results for subsequent references. Care must, of course, be taken not to use old data items that are no longer valid. One common approach to this problem is to create a large data base containing all of the derivable information, and to keep this constantly up to date by making all of the necessary modifications whenever anything is changed. This method works well for very simple problem domains, but the data base becomes ridiculously large when used with a system employing a large variety of types of knowledge. In such a system every small change would have a large number of consequences, very few of which would ever be referenced. Much computation would be wasted in deriving all of this useless information.

Since BUILD is intended to be a sort of prototype for a system employing a great deal of knowledge about the world, I have not used the complete-data-base method described above. Instead, I use a scheme in which secondary data is only derived when it is required, but is then saved for

subsequent references as items and properties in CONNIVER's pseudo-associative data base. For example, consider a program that needs to know whether block A touches block B, and, if so, at what points. BUILD will look first in the data base to see if it already knows that A touches B. If this succeeds, it will return the information about points of contact stored in the same data item. If, instead, the search fails, BUILD will see whether it knows that A and B are not touching. Failing this, it will look for equivalent forms such as B TOUCHES A. Only if all these searches fail will BUILD invoke the touch testing routine, which does the necessary work required to find the answer. In addition to returning the answer, this routine stores it away as a new data item so that the search phase will succeed the next time this information is required. If block A is ever moved, all of the touch information involving A, and any other secondary information that depends on A's position, will be erased by the system demons (if-added and if-removed methods).

This system has several advantages: No computation is ever made when it is not needed, and no computation is needlessly repeated. Great flexibility is available to the programs in this system, because functions can be called from anywhere, without regard for whether the proper data

base environment has been set up. If storage becomes crowded with items of secondary data, any or all of these items can be erased, resulting only in degraded performance, not in disaster. The programmer is freed from continual decisions about whether a given datum is useful enough to save.

Several extensions to this idea, not currently implemented, suggest themselves. The routines that create these secondary data items could note (on the datum's property list, perhaps) how hard this particular item was to derive. In the touch test, for instance, it is harder to tell whether blocks are touching if they are very close than if they are at opposite ends of the table. When the time comes to throw out some of these secondary data items, those that were won most easily will be the first to go. Another useful item to save with a datum is some rough heuristic estimate of how useful the information is expected to be in the future. This, too, can be used by the system in deciding what to forget. Even if this heuristic estimate is wrong, it can only slightly degrade performance; as long as the heuristics do better than chance, overall efficiency will be gained.

This type of memory management is not entirely new, though I know of no other modeling system which uses it

nearly this extensively. (Winograd makes token use of a similar system in his CLEARTOP predicate.) Some sort of list-structured associative memory, such as PLANNER and CONNIVER provide, seems almost essential for the storage of secondary data items. Without this facility, one would be almost forced to use cumbersome fixed-length tables to store the data for easy retrieval and this, in turn, would be a strong inducement to return to fixed, complete data bases. The GOAL function in PLANNER, which first checks the data base for assertions of the desired form, then resorts to theorems to achieve the goal, was intended for the type of memory management I have described but, at least in its MICRO-PLANNER form, it is too inflexible to be of much help. This will be discussed in section 5.1.

2.3 Format of 3-D Models

BUILD's world consists of a table (of any specified dimensions) and a population of movable objects called blocks. These blocks can be any shape which BUILD recognizes (currently bricks and right-triangle wedges) and can have any dimensions. They can be in any spatial position and rotation. The specification of the model must be complete; dealing with uncertainties of size and position

is a problem for another thesis (see, for instance, <BOERG 1972>). BUILD must be able to store several world-models or states at once, without confusing them. It might, for instance, have models of the world's present state, the goal state toward which it is working, some hypothetical proposed state which is being investigated, and a list of the states it has passed through in the course of its solution. This was a problem in MICRO-PLANNER, but the context structure of the CONNIVER data base provides exactly the facilities needed. It is not too expensive to save many slightly different contexts because shared information is not recopied in each context. In this thesis the words "state", "context", and "scene" are used more or less interchangeably.

Blocks exist independently of states. A given block may appear in several states or in none at all, and may occupy a different position in each state. A block will have the same size and shape wherever it appears; these properties are placed in the data base when the block is defined. Other invariant properties, such as color, could be treated likewise. The size of a block is a vector of parameters whose length and interpretation depends upon the block's shape. For bricks the size vector has length 3 and is simply the three dimensions in any order. For wedges the

size vector again has length 3 and specifies the two legs of the triangle and the thickness. More complicated blocks will require that more parameters be specified. In this system it never makes sense to talk about size except in relation to a particular shape.

Points are simply represented by their 3-space coordinates (x y z). Planes are represented by 4-tuples of the form (x y z w), where (x y z) is the unit normal vector to the plane and w is the distance along that vector from the origin to the plane. Each plane divides space into two half-spaces, which I will call the inside and the outside; the normal vector points toward the outside. A point (A B C) lies on the plane (x y z w) if AX+BY+CZ-W=0. If this quantity is greater than zero, the point is outside; if less it is inside. This quick test will be useful in the touch testing systems.

Generalized positions of objects (position and rotation) are indicated by 4x3 arrays called AT arrays. These are really equivalent to homogeneous coordinate arrays, except that the right-hand column is chopped off. In the absence of scaling transformations this column is always 0 0 0 1, so no information would be gained by keeping it. The first three rows of the array consist of a 3x3 rotation matrix, while the last row simply contains the x, y, and z displacements of the object. To find the real 3-D

position of a point whose position relative to a block is known, one simply multiplies the given (X Y Z) vector by the upper 3x3 matrix, then returns the vector sum of the result and the bottom row of the array. These AT arrays can be thought of as mappings from one set of spatial coordinates to another, usually from the block's coordinates to the "real world". It is possible, however, to have one block's AT array relate to the space defined by another block, so that when the second block is moved, the first moves also.

When it becomes necessary to generate the vertex positions or face plane equations of a block in some state, the necessary formulas are retrieved from among the properties of the shape name. These are LISP functions which generate all of the vertex coordinates and face plane coefficients using parameters supplied by the block's size property. These are generated for a block at the origin and unrotated, and are then converted to their true values using the block's current AT array for the state in question. As described in section 2.2, these final values are saved for future reference. The shape also has properties giving formulas for weight, center of gravity, and maximum distance of any point on the block from the "center point". This last quantity provides a quick way for the touch test to eliminate distant objects. Also among each shape's

properties is a wealth of topological information: which vertices and lines surround each face, which faces meet at each vertex, which faces are adjacent, and so on.

2.4 Display Routines

In order to present BUILD's activities and plans in an intelligible form, I have written routines which are able to display any state generated by BUILD on the system CRT, and to produce hard copy on the X-Y plotter. These routines make extensive use of the LISP display utilities provided by Jerome Lerman and Jon White.

Each display is created relative to some viewpoint. A viewpoint is a list containing an AT array, a scale factor, and an indicator showing whether the display is to be an orthogonal or a perspective view. The AT array gives the position and orientation of the eye, which can be turned and moved just as though it were a block. The scale factor controls the size of the image and the extent of the field of view, like the focal length of a camera lens. The endpoints of each line are mapped into "eye space" using the eye's AT array. The resulting X and Z coordinates, times the scale factor, become the horizontal and vertical coordinates of the points on the scope. If perspective is

to be used, these values are first divided by the Y coordinate of the point in eye space; i.e. the distance of the point from the plane of the eye pupil.

The system currently eliminates only those hidden lines which are formed by the meeting of two faces, both facing away from the eye. Thus single convex objects appear correctly, but objects behind them will show through. With some extra effort a true hidden line algorithm could be implemented, along with a module to generate shadow lines, given an arbitrary position of the light source. This could be of great use to the vision system by providing a sort of feedback loop. For any proposed 3-D model of a scene, whether arrived at by deduction or guesswork, the display system could generate the corresponding line drawing. This could be compared with the data coming from the eye, and on this basis the model could be accepted, rejected, or altered slightly and tested again.

Chapter 3: The Touch Test

3.1 General Requirements

For obvious reasons, BUILD requires a means for determining whether two blocks are touching, not touching, or colliding (trying to occupy the same volume of space). The handling of touch information, once found, is described in Section 2.2; this section will be devoted to the test itself, and the various alternative methods that were considered. This test ought to accomodate the objects of arbitrary shape, concave or convex, that we would like eventually to use, and, since it must sometimes be used repetitively while an object is being moved, should be fairly fast. Since the floating-point numbers used by the modeling system exhibit roundoff error, touching must be defined in terms of a programmer-set tolerance: Two objects touch if they approach within this tolerance or overlap by less than it.

Whenever two objects are touching, additional information is required for use by the stability test. A list must be created giving the 3-space coordinates of all points of contact between the objects. If the contact is made along a line segment or over an area, the endpoints or

vertices are returned and treated as the sole points of contact. (While this is sufficient information for stability testing, more data about the area of contact would be necessary if the system were to use glue in its constructions.) The touch test must also report the normal vector at each point of contact. This is just the normal of a plane separating the two objects at this point. Without this information it would be impossible for the stability test to separate the normal and frictional components of a force. For two convex objects a single normal vector is adequate, since a single plane must pass through all points of contact.

3.2 The Recursive Test

Gerald Sussman has described two methods for collision-testing and has studied the efficiency of these methods for randomly-generated 3-dimensional bricks <SUSSMAN 1971>. Sussman only considered collision detection, not the problem of characterizing the points of contact. His first method, which he calls the recursive test, generates inscribed and circumscribed spheres around each brick. If the circumscribed spheres do not intersect, the blocks are known to be clear of one another. If the inscribed spheres

intersect, the blocks are known to be colliding. If neither is the case, each brick is divided into two parts and the test is recursively applied to each part.

This method works very quickly for blocks very far apart or severely overlapping, but slows down drastically for blocks that are almost touching. If two blocks are exactly touching over some area, a large number of recursive branches will have to run until the spheres reach the system's touch-tolerance level in size, a very costly process. Since most blocks in a BUILD state touch at least one other block, this method is clearly not appropriate. In addition, this method works only on standard shapes. The basic unit could be a tetrahedron instead of a brick, and all polyhedral objects could be divided into collections of these, but this is a very difficult and costly process.

3.3 The Simplex Test

Sussman's second technique, useful for blocks of any convex shape, makes use of a form of the Simplex technique from linear programming. (For a full discussion of this technique, see <ZUKHOVITSKY, AVDEYEVA 1966>.) Each block is represented by a set of linear inequalities in X, Y, and Z, corresponding to the faces of the block. A point is inside

the block if it satisfies all of the inequalities defining that block. If the two blocks in question are colliding, there will be some point that lies within both blocks, and which must thus satisfy all of the inequalities of both. Finding such a point, or determining that no such point exists, is a common problem in linear programming and well developed techniques exist for systematically searching space for this point. The inequalities are represented in a matrix (12 x 4 for two bricks) and carefully selected row substitutions are performed, typically about four in number, until the answer is reached. To allow for the roundoff tolerance, this operation must be performed twice, once with each plane moved outward by the tolerance value, and again with each plane moved inward by the same amount. Blocks are declared to be touching if they intersect while expanded, but not while contracted.

There are problems with this method also. Concave objects must be broken up into convex parts before this test is used. Furthermore, it is quite hard to extract the necessary point-of-contact information using this test. One point of contact is returned as the point which satisfies all of the inequalities, but I could find no good way to extract the other points or the normal vectors.

3.4 The Line-Face Test

The linear programming test was implemented but, because of the problems noted above, this method soon gave way to one more geometric in flavor. This geometric test operates by checking each edge line of one object against each face of the other object, and vice versa. If any line is found to penetrate a face, the objects are colliding. This line-face test is faster than might be supposed. A quick check is first run to see whether the objects are too far apart to be touching, using what amounts to circumscribed spheres. If the objects are close enough, the line-face testing proceeds. Most of the line-face tests return at once because both endpoints of the line are found to be on the same side of the face plane. If this is not the case, the intersection of the line and the face plane is found, and the border of the face itself is traced out to determine whether this point is inside the border and thus on the face. Because of the two levels of pre-testing, very few of these costly border tracings are performed in analyzing an entire scene.

There are a number of exceptional situations which might be found during the line-face testing, and each of these requires its own exception-handling routine. Such

exceptions occur when one of the endpoints of a line is found to lie on a face (within the touch tolerance, of course) or when a line grazes the boundary of a face along an edge or at a vertex. These exceptional points usually either indicate a collision or a point of contact, according to the local geometry. Whenever an exception handler finds a contact point, it computes the appropriate normal vector and adds both items to a list. A certain amount of bookkeeping is required to keep the programs from investigating these points several times, once for each line and plane involved.

This test has several advantages. Unlike the Simplex test, it produces a complete list of touch points and normal vectors. In addition, it can operate directly on blocks with concavities without first dividing them into convex pieces. It is, however, quite a large and complicated system and runs rather slowly, despite its pre-testing. In compiled LISP form, the test took a fairly constant 18 to 20 seconds of CPU time for pairs of nearly-touching bricks. Touching bricks took slightly longer, while colliding bricks took around 3-5 seconds. This compares with about three seconds for the uncompiled Simplex test. These times, of course, are extremely sensitive to machine speed and programming details; I include them only to give the reader

some rough idea of how these algorithms compare in efficiency.

3.5 The Separating Plane Test

The touch test currently used in BUILD was stumbled upon by the author while trying to improve one of the exception-handling modules of the line face test. This new test works only for convex objects, but makes up for this limitation by being very much faster than the older line face test. The new test depends upon the observation that if two convex objects can be separated at all, they can be separated by a plane (not necessarily unique) through two vertices of one object and one vertex of the other. Either block may be selected to contribute the two vertices, since a solution will exist in either case. The separating plane must pass between the centers of gravity of the two blocks, and all of the vertices of each block must be either on the plane (within the system's roundoff tolerance level) or on the same side as the block's center of gravity.

Given these constraints that must be met by any separating plane, it would be possible to find if such a plane exists by conducting an exhaustive search. This search would match each pair of vertices from one block with

every vertex of the other, construct the plane through these three points, and then check whether the centers of gravity and vertices are indeed separated by this plane. As mentioned in section 2.3, these point-plane comparisons are quick, requiring only 3 multiplies, 4 adds, and a compare for each. If a good separating plane is found by this process, it is relatively easy to find the contact points; only those vertices and edges that lie on this plane need to be checked for coincidence. The normal vector at all of these contact points is just the normal vector to the separating plane. If the objects are not touching at all, there will be a separating plane but no contact points. If the exhaustive search is unable to find a separating plane, the objects are colliding.

In fact, much of this exhaustive search can be avoided by using a more efficient search strategy. First, the system finds the two vertices of the first block, P1, which are the closest to the center of the second block, P2. The vertices of P2 are then scanned in the order of their closeness to P1's center until one is found which, along with the two P1 points, forms a plane cutting between the two centers of gravity. Usually this first plane is the desired separator, with no vertices of either block on the wrong side of it. If not, only the offside vertices, along

with the initial three, need to be considered in the subsequent search.

This test is much faster than the line-face test. For distant objects, the same maximum distance cutoff is used as for the older test. For touching or nearly touching bricks, the new test averages about..4 seconds to find the separating plane, with an additonal second or so being required to determine all the contact points. These times are for the compiled LISP version. The new test does somewhat worse with colliding blocks, since it tries very hard to find a separating plane. In the worst cases it takes about 1.5 seconds to determine collision.

In theory, this test is very similar to the Simplex test described above. Instead of looking for a point that simultaneously satisfies a set of linear constraints representing the plane faces of the objects, the new test looks for a plane to fit the linear constraints supplied by the object vertices. Each of these points must end up on the proper side of the plane. The W, X, Y, and Z of the plane are the unknowns. The test could have been reprogrammed to use the Simplex method on a matrix, but after some consideration I decided that the current form of the program is probably faster for all but the most complex objects, due to the overhead cost of setting up the matrix.

To use this test with concave objects, it is first necessary to divide them into convex pieces. One method for doing this is to extend planes through concave edges until the plane hits another plane from the inside. A two-dimensional illustration of this is given in figure 3-1.

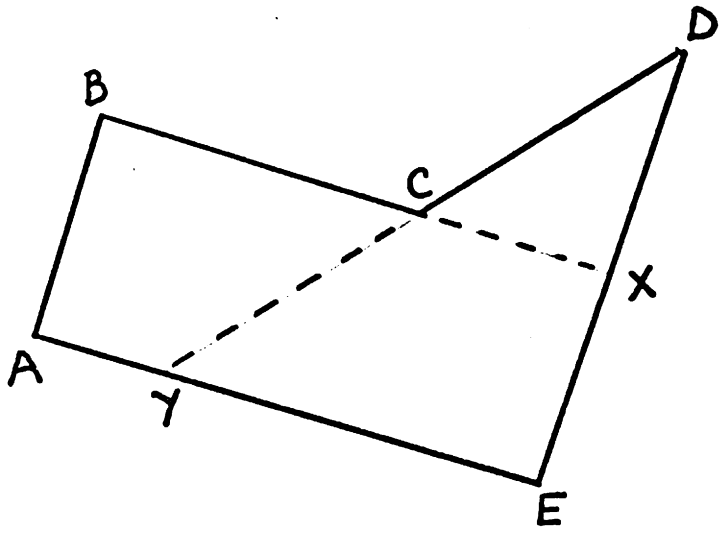


figure 3-1

Line B–C is extended through the concavity at C until it intersects line D–E at point X. D–C is likewise extended to point Y. This process forms two maximal convex objects A–B–X–E and D–Y–E. It does not matter that these overlap, since parts of the same object will never be tested against one another. This object divider is not currently implemented.

3.6 The FINDSPACE Proposer

The FINDSPACE proposer is not really a part of the touch test, but is intimately related to it, so I include a description of it here. Whenever BUILD needs to find an arbitrary place to put some block, it calls FINDSPACE, which searches around the table for a place where the block in question will fit. This is necessary whenever a block has to be set aside, either because it is in the way, or because it is resting on some other block which must be moved. The table is assumed to be large enough to hold all of the blocks with no trouble, so FINDSPACE never tries to pile blocks on top of other blocks.

FINDSPACE consists of two major parts, the proposer and the tester. The proposer suggests various new locations for the block in question, and the tester sees if the block can be placed there without hitting other blocks. The tester is, of course, simply the touch and collision testing module described above. The current FINDSPACE proposer is quite simple, but works adequately well if the table is not too crowded. First, the faces of the block are scanned to find the smallest one upon which the block can stand and still be reasonably stable. The proper rotation and height for the block are computed, in order to bring this face into

contact with the table. A coarse scan of the table is then initiated, using fixed increments of X and Y. The block is imagined at each of these X,Y points and the touch test is run. This continues until a good position is found, or until the scan is completed, in which case FINDSPACE gives up. Some ideas for a better proposer are discussed in Chapter 7.

The current proposer has two other features which are of use to BUILD. First, any location produced by FINDSPACE can be rejected by the caller and returned for a new value. FINDSPACE is simply called again with the old value as an optional extra argument. The X-Y scan is re-initiated from this point. Also, FINDSPACE can optionally receive a list of other contexts whose blocks are to be avoided just as if they were present in the current context. The normal use for this feature is to prevent FINDSPACE from placing blocks in locations that are earmarked for later occupation by blocks in the goal state.

Chapter 4: The Stability Test

4.1 General Requirements

The heart of BUILD is the stability test, the module that looks at a state of the blocks world and decides whether anything is going to fall. Since BUILD does not have the real-world feedback that humans use so extensively, it cannot start to release a block and grab it again if it begins to fall. All questions of stability must be decided correctly by the stability test if BUILD's plans are to succeed. Since, to make BUILD's problem domain interesting and realistic, friction was included in the world model, the stability test has to deal with cases where frictional forces occur.

A construction planning system cannot make intelligent choices if its stability testing module provides only yes or no answers. True, it is possible to try placing the various blocks in position in some random order, using the stability predicate to determine which of the moves are legal. In many very simple cases this process will produce a legal plan, though not in a very efficient manner. When, however, the situation requires that more complex methods be used, BUILD must be able to gain some understanding of the nature

of the difficulty it has encountered, in order to know what solutions to try. This need arises even in such a simple case as a block buried by other blocks. BUILD tries to move this block, but the stability predicate complains. Obviously some other block or blocks must be moved away first, but which ones? Since the stability predicate is unable to report which blocks are falling, the planning programs must engage in a sort of "Twenty Questions" dialogue with the stability test in order to determine the culprits. This is clearly ridiculous; it is even more ridiculous to attempt to find movable sub-assemblies or to position temporary supports under such conditions.

4.2 The Blum-Griffith-Neumann Test

In 1970, a short time before the start of my own research, Blum, Griffith, and Neumann <BLUM, GRIFFITH, NEUMANN 1970> described an algorithm that essentially solved the problem of stability testing in cases with friction. Unfortunately, this test was of exactly the yes/no type that I have been condemning and, because of this, it proved to be useless for my purposes. Their method makes use of linear programming methods very similar to those used in the touch test described in section 3.3, but on a much larger scale.

The unknowns in this case are the X, Y, and Z components of the forces at each contact point in the scene. For each block, six equations are derived: The sum of the forces on the block, including gravity, must equal zero in the X, Y, and Z directions, and the X, Y, and Z moments must also equal zero. More constraints, in the form of inequalities, are derived from conditions at each point of contact: The normal force component must be zero or positive, since a negative force would mean that something (glue, perhaps, or chewing gum) is holding the blocks together. Also, the frictional force at a point can be no larger than some constant, MU, times the normal force at that point. All of these equations and inequalities are represented by their coefficients in a large matrix. Row substitutions are then performed to eliminate the equalities. Additional row substitutions are performed until either all of the inequalities are satisfied, indicating a stable state, or some inequality is found which cannot be satisfied, indicating an unstable state.

For a while I was unable to devise a better algorithm, so I implemented this test in the hope that the necessary additional information could be extracted from the remains of the coefficient matrix. This proved to be a forlorn hope. When the state was unstable, the inequality that

ultimately failed often bore no decipherable relation to any of the falling blocks. When the state was stable, the set of returned forces was full of spurious entries. Forces appeared at random contact points for no reason at all, only to be balanced by frictional forces that would otherwise not have existed. In addition to these problems, the test used a huge amount of core and was very slow. If B is the number of blocks in a state and P is the number of contact points, the coefficient array has 6B+2P rows and 3P columns, and at least 6B complete passes over the array are required. For a scene with ten bricks, this means that at least 60 passes will be made over an array with about 170,000 entries. This is true whether the bricks are piled in a complex structure or are sitting in a row on the table. Because of all these problems, I was forced to abandon this test in favor of a more heuristic approach.

4.3 The Heuristic Test

In calling my current stability test heuristic, I do not mean to imply that it is approximate or that it is prone to incorrect conclusions. I use the term merely to indicate that the flow of control in this test is flexible and data-dependent, at least compared to the methodical matrix

crunching of the older test. While the system of inter-block forces in any given structure arises from a simultaneous attempt of all the blocks to balance the gravitational forces acting upon them, it does not necessarily follow that a stability testing program must be based upon simultaneous equations. A consideration of any assortment of real block structures will show that in most cases the inter-block force relations fall into clear chains of cause and effect: Block A, acted upon by gravity, exerts certain forces at some set of points on block B below it; block B, acted upon both by gravity and the force from A, exerts greater downward forces on C; and so on. These simple causal chains of forces can be determined on a block-by-block basis, without resorting to simultaneous solution methods. Of course there are cases of equilibrium or mutual support which do require a simultaneous solution, but these can be handled by deriving and solving a very small local set of simultaneous equations or, as is actually done in the current test, iterating to a solution.

It might, at first glance, seem that this approach merely substitutes a slow and roundabout method for a fast direct one. In fact, however, the new test has a rather dramatic speed advantage in all but the most perverse scenes. The reason for this is that the new test only has

to deal with the forces and imbalances that actually appear in the scene at hand—not with all potential motions and interactions, the number of which can be immense. If a block is acted upon only by the downward force of gravity, for instance, the new system will never have to worry about the constraints on sideways motion or on rotation around a vertical axis. The problem just doesn't arise. Actual performance figures will be discussed later.

Preserving the causality structure of the problem yields other benefits as well, and these are far more important than mere speed. With the new test it is possible to determine not only that a force exists between two blocks, but also which block is pushing and which is being pushed, or, in other words, which block is supporting the other. This information is obviously useful in deciding which block to place or remove first. Similarly, if a block is found to have an uncorrectable imbalance of forces, this is a clear indication that the block will fall in whatever way these forces dictate. The blocks pushing on this block will then have imbalances of their own, and so on. Any instability in a structure is thus clearly decipherable as to type, location, and cause, rather than appearing as an inability to satisfy some frictional constraint on a block far from the scene of the disaster. The availability of

such information makes it much easier to find ways to correct the instability.

Whenever it is determined that one object, B1, is exerting a force on another object, B2, an item of the form (B1 SUP-BY B2 (P F) (P F) ... .) is added to the data base. SUP-BY is short for supported-by. The (P F) pairs indicate the points of contact where the pressure is being exerted and the force vector at each of these points. If B1 also presses on another block, say B3, then (B1 SUP-BY B3 . . .) is added as well. The absence of any (B1 SUP-BY. . . .) items in a context indicates that B1 has not yet been considered by the stability test, and that its supports are, at present, unknown. In the rare case that a block doesn't need any supports, either because it is weightless or because its weight is exactly balanced by incoming forces, a dummy item of the form (B1 SUP-BY NIL) is added.

The stability test begins by creating a list of all the blocks in the scene whose supports are unknown. One by one, in some random order, these blocks are removed from the unknown list and passed to a function called CHECKSUP. This function gathers together all the forces known at the time to be acting on the block, including the block's own weight and the forces applied by other blocks as indicated by the SUP-BY items. From these forces it computes the net force

acting on the block and the net rotational moment around its center of gravity. Next, a list is created of all the points of contact between the block in question and neighboring objects. CHECKSUP scans down this list, looking for a point where it can apply an outgoing force that counteracts at least part of the net force or moment or both. Each applied force must, of course, be legal by local criteria: The normal component of the force must be positive (pushing), and the frictional component must be less than the normal force times the coefficient of friction. When such a point is found, the applied force is recorded, a new net force and moment are computed around this point, and the whole process is repeated. Whenever CHECKSUP has a choice of points that can accept a force, it prefers to put the force all in one place instead of dividing it between points, and it prefers to create normal forces rather than frictional ones. Whenever possible, it will correct an imbalance by removing an old force rather than by creating a new one.

CHECKSUP iterates until either the net force and moment are completely eliminated, indicating a stable block, or until they can be reduced no farther. In the latter case the block is deemed unstable and is place on a list of losing blocks. With it is stored the remaining net force

and moment, and the pivot point around which the block is expected to topple (unless the imbalance is in force only, indicating a translational motion). This will sometimes result in a new force being applied to an object that has already been tested by CHECKSUP. Such a block must be returned to the unknown list to be tested again. This is true even if the block is on the loser list, since the new force might be exactly what is needed to balance the block and make it stable. Forces applied to the table or to blocks declared to be immovable (glued down) do not result in new unknowns, nor does a block return to the unknown list if the new force added is smaller than the system's roundoff tolerance level. Infinite looping only occurs in certain rare and contrived cases in which roundoff errors, compounded by long lever arms, create spurious forces which keep pushing the program away from the point of convergence.

When no blocks remain in unknown status, the loser list is inspected. If it is empty, the stability test succeeds, leaving all of the SUP-BY items in the data base. If there are losing blocks, the blocks that are pushing on the losers are checked to see if they could be supported by something else instead. In figure 4-1, for example, it could have been decided that B is supported by A and D, and that D, unable to support this force, was about to topple in a

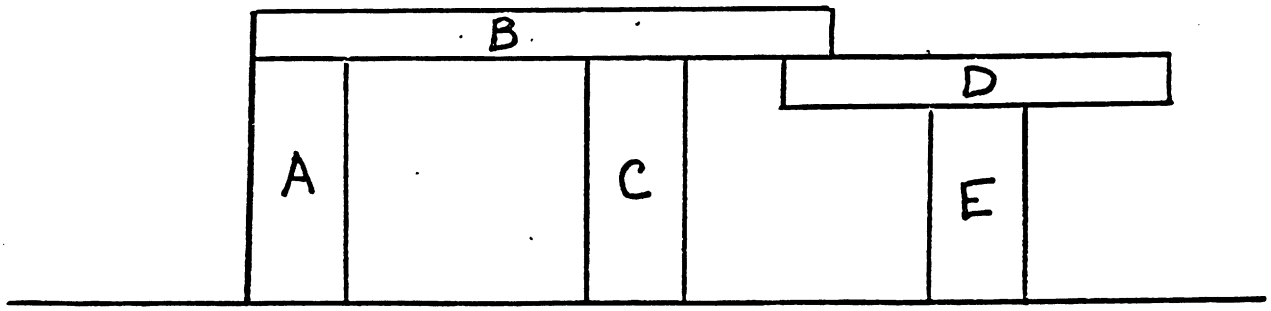


figure 4-1

counter-clockwise direction around the upper-left corner of E. Before returning, the system checks B to see if another block could absorb some or all of the force being exerted on D. The alternative support provided by C is found, the force is applied there instead, and everything succeeds. If blocks remain on the loser list after this process, the stability test fails and returns the list. All of the losing blocks are thus reported, along with specific information as to which way each one will move or pivot.

A detailed example will perhaps bring the entire testing process into better focus. Consider the structure shown in figure 4-2. We will assume that the coefficient of

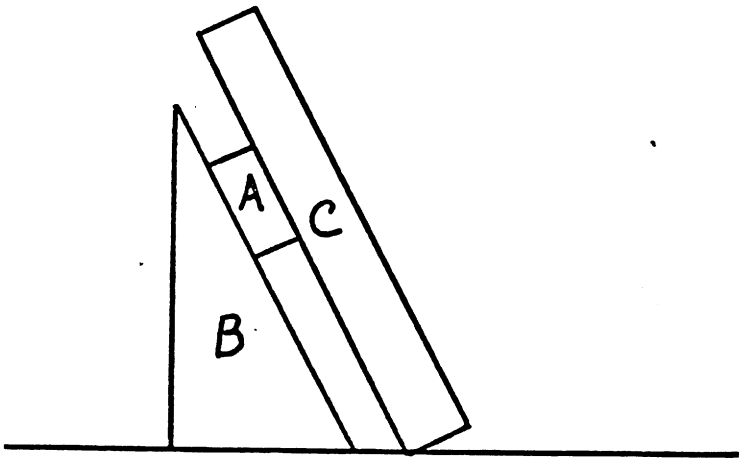


figure 4-2

friction is such that A would normally slide down the face of B, but that the added force from C is enough to hold it. The structure is therefore stable; the steps that the stability test follows in determining this are given below. Remember that the order in which unknown blocks are considered is random, so this represents only one possible sequence.

UNKNOWNS = (A B C)

1. CHECKSUP first considers block A. Since there are no known external forces acting on A, only the force of its own weight is considered. A is found to be pressing down and to the left on B. Some of the down-right sliding force of A is passed on to B as

frictional force, but not enough to prevent A from sliding. Block C doesn't help at all, since the motion of A is along its surface and there is no known normal force between the two. (A SUP-BY B . . .) is added, for the force that was found, and A is placed on the loser list, with an indication that it will slide down and right.

UNKNOWNS = (B C)

2. B is considered next, taking into account both its own weight and the applied force from A. It is found to be pressing down and very slightly to the left against the table, and to be stable. (B SUP-BY TABLE . . .) is added.

UNKNOWNS = (C)

3. C is tested next. CHECKSUP determines that it is pressing down and to the left against A and down and slightly to the right against the table. The rightward frictional force against the table is small enough compared to the downward normal force to be passed on to the table with no problems, so the block is stable. (C SUP-BY TABLE) and (C SUP-BY A) are added to the data base, and because of this newly applied force A is removed from the loser list and returned to the unknown list.

UNKNOWNS = (A)

4. A is re-tested, this time under the applied force from C, as well as its own weight. Again it is pressing down and left against B, but this time the force is greater. This means that B can now accept all of the down-right frictional force, so A is now stable. The old (A SUP-BY B . . .) item is deleted and a new one, indicating the new greater force, is added in its place. Since the new and old applied forces are significantly different, B must be placed on the unknown list for re-testing.

UNKNOWNS = (B)

5. B is now re-checked and is again found to be stable, pressing down and slightly left against the table, but with a different force than before. A new

(B SUP-BY TABLE . . .) item is added in place of the old one.

UNKNOWNS = NIL

6. There are no more unknown blocks, and the loser list is empty. The stability test thus returns with an indication that the structure is stable. The data base is left with the following items in it:

(C SUP-BY A . . .)
(C SUP-BY TABLE . . .)
(A SUP-BY B . . .)
(B SUP-BY TABLE)

This example is typical of most normal structures in that the solution was arrived at more or less directly, without the necessity for looping and slow convergence to some equilibrium of inter-block forces. There is, of course, some looping and shifting around of forces within each call to CHECKSUP, but this is of a very local nature and is therefore not too costly. The number of CHECKSUP calls required for a given structure depends on the system's luck (or skill) in arranging blocks on the unknown list. If, in the above example, the initial unknown list had been (C A B) instead of (A B C), only three calls to CHECKSUP would have been made instead of five. The optimal strategy is for the test to work its way down the tree of support relations; that is, whenever A is supported by B, A should be considered before B. Unfortunately, these support relations are what the stability test is trying to find, so

they are not available for ordering the initial list. Remarkably good efficiency is, however, obtainable by the simple expedient of sorting the unknown blocks by the height of their centers from the table, and considering the higher blocks first.

Despite all efforts to minimize the need for iterative convergence on a solution, there are some cases where such iteration is inevitable, usually because the inter-block forces are in some sort of equilibrium or act upon each other in a loop. In figure 4-3, for instance, block A might

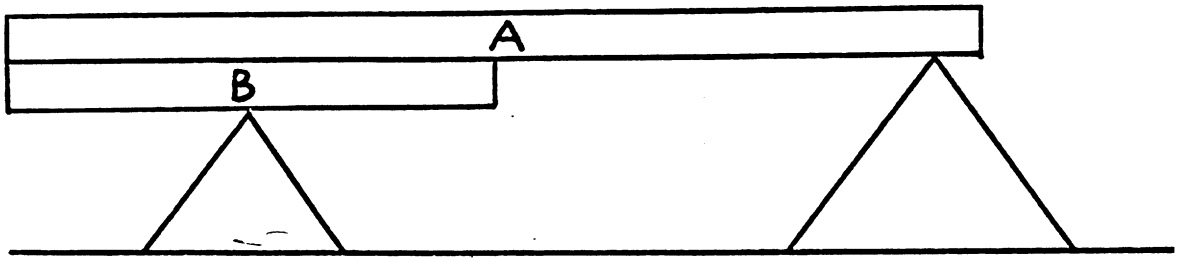


figure 4-3

first try to distribute its weight equally at both ends. Block B, receiving a force on its left end, pivots and exerts an equal upward force at its right end. A is rechecked and, in view of the new upward force at its center, reduces the force at both ends. Block B is checked again and, with less force applied to its left end, it reduces the upward force it is exerting at the center of A. A, in response, applies a little more force at both ends, and so on. This is not an infinite loop; gradually the inter-block forces are converging on the steady-state solution, which in this case is 1/3 of the force applied at each end and the center. As this value is neared, one of the applied forces will differ from its previous value by less than some small tolerance and the looping will be halted. If this tolerance is set at .0001 of block A's weight, about 13 iterations will be required, a slow but not impossible process. Figure 4-4 shows a situation where the support relations form a loop: A supports B, B supports C, and C supports A. In this case the testing program would have to cycle around the loop several times, but, as above, the system converges to an answer eventually. It would be possible for the system to recognize when it is looping, derive the equations governing the loop (usually few in number and very simple) and solve directly for the final

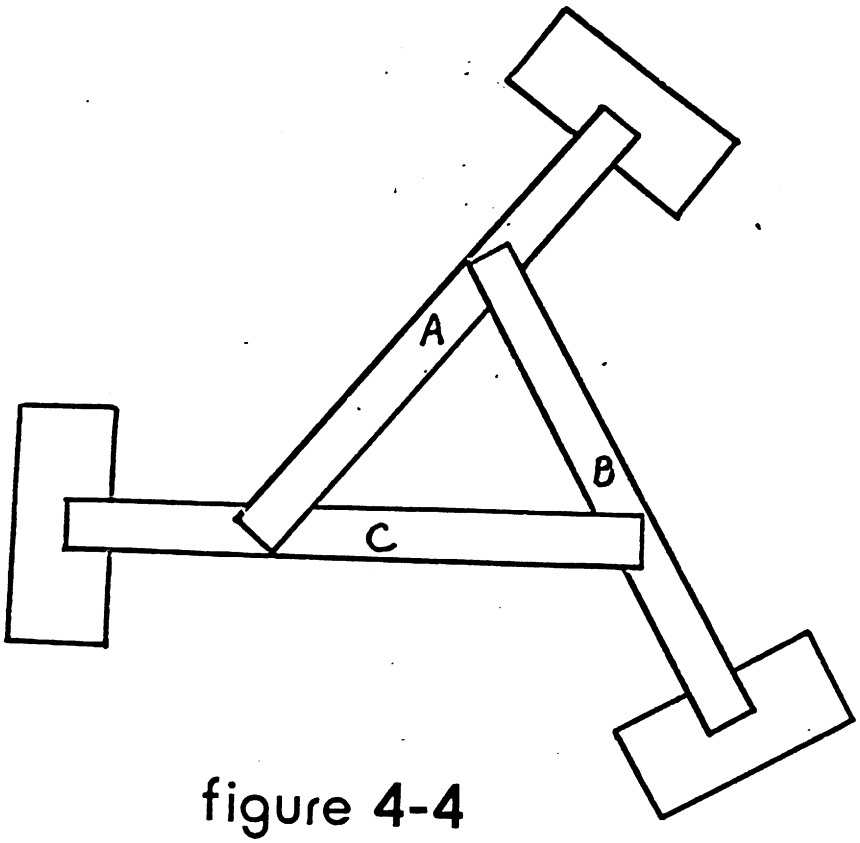
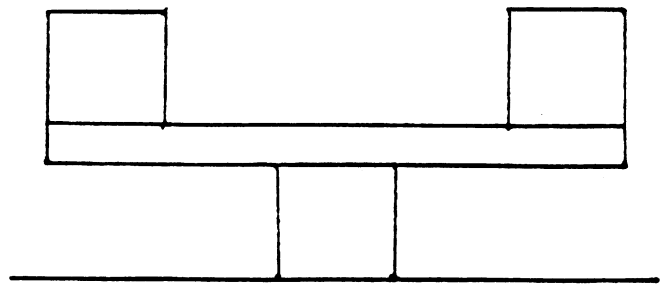


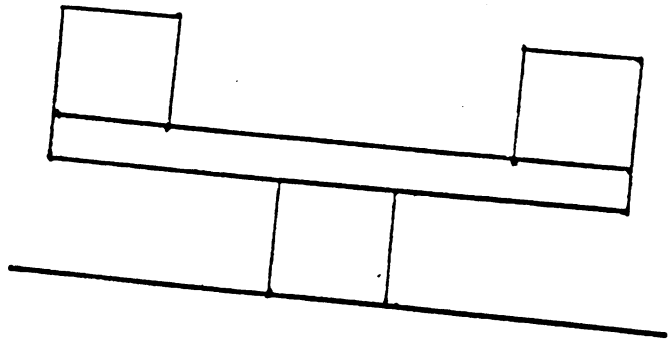
figure 4-4
(seen from above)

forces, but since the looping cases are not common and speed is not a main goal of my research, I have not worked on this.

The speed of this test, while far from optimal in the above cases, is still quite respectable overall. Figure 4-5 shows a number of scenes, with the CPU time required to test each. The figures given are for the compiled LISP version of the test, as run on the PDP-10, and they do not include the time required to find the touch relations in the scene. I



2.7 sec



9.6 sec

7.9 sec

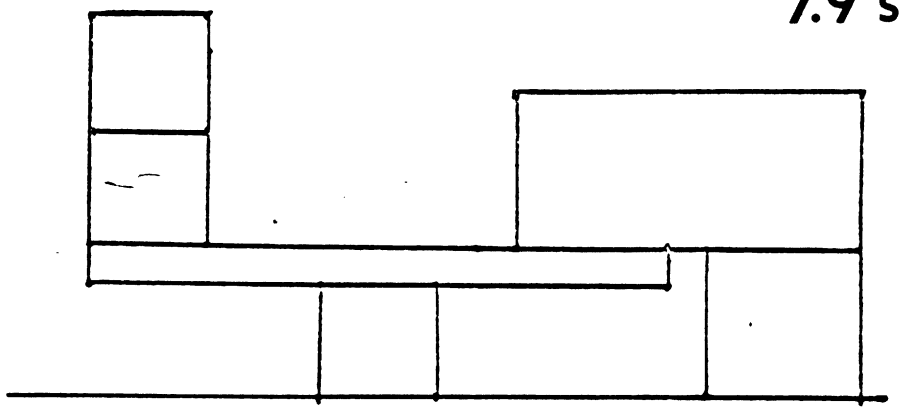


figure 4-5

do not, unfortunately, have comparable times for the Blum-Griffith-Neumann test, but I can report that I spent over two hours one afternoon waiting for the test to finish a four-block seesaw, a period of time which must certainly have corresponded to 10 or 15 minutes of CPU time. When I stopped the test, it was about half completed. As far as I have been able to determine, my test handles correctly all of the structures that the other test does, except for some obscure cases that are bothered by roundoff error, but since my test is much more complicated than the older one it is much harder to feel confident that all possible cases have been dealt with. It is clear that it works well for the relatively tame scenes that are encountered in typical block-building, and for several hard test cases as well. It should be noted in this regard that there exists a class of structures whose stability is indeterminate from considerations of macroscopic geometry alone. Figure 4-6 shows one such case. The inward force of the grippers will be applied either to A or to B or to both. If only one block receives this force, the other will fall, but if both receive half of it the scene is stable. In the real world, the issue is decided by microscopic imperfections on the block faces, and by deformation of the materials at the surface of contact. The old test is always optimistic for

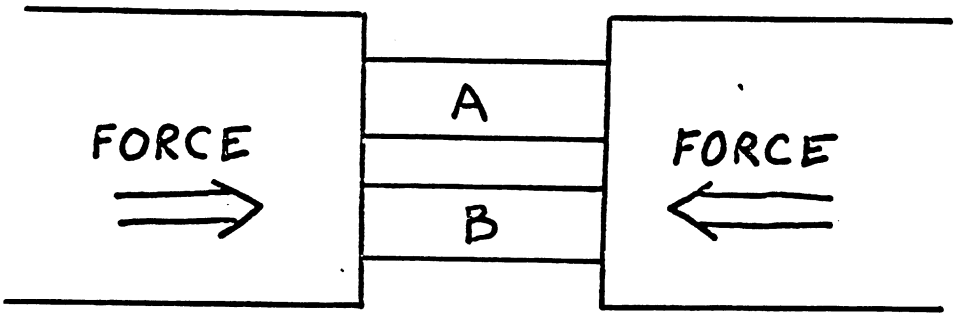


figure 4-6

stability; my test is likely to be arbitrary, usually in the direction of pessimism.

It should be noted that the tree of support relations represented by the SUP-BY items that the test returns may not be unique. To be perfectly general, a block's supports can only be expressed as an OR condition of sets of other blocks. For example, in figure 4-7, A's supports are (C B E) or (C D E) or (C D F). If any of these sets are all present, A will be stable. To find these sets a stability test would have to try essentially all of the subsets of a scene. Since this is not done, it is not really the case that a block cannot be placed until its known supports are

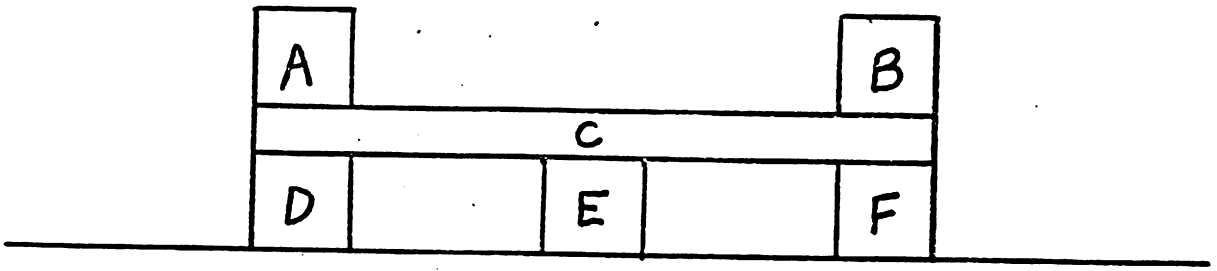


figure 4 -7

in place, since other supports might serve as well. Nevertheless, this rule is a useful planning heuristic which can be abandoned if trouble arises. Similarly, it is not always true that a block can be placed if its known supports are in place. A block might also depend upon the forces from the blocks it supports if it is to be stable. Figure 4-8 shows such a case: The data base will show only (A sup-by B . . .), (B SUP-BY C . ...), and (C SUP-BY TABLE . . .), but B depends on the presence of A as well. This is really a case of mutual support, with A and B depending on each other. Usually the data base will show support only in one direction in such cases. If the system ever finds an

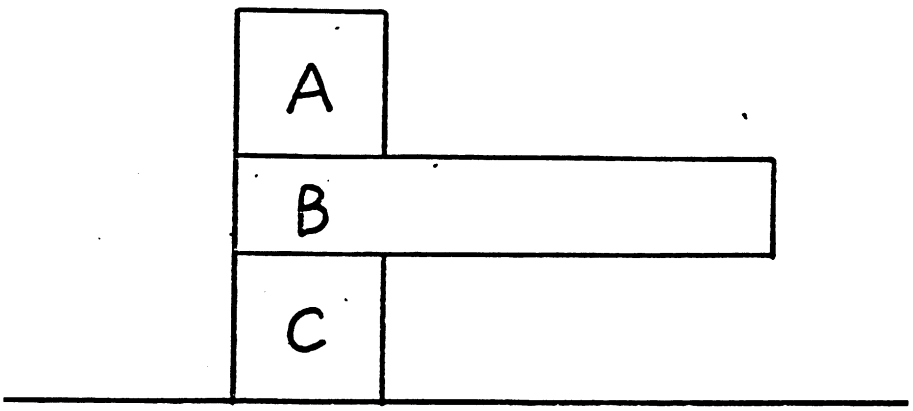


figure 4-8

instability resulting from the placement of a block whose known supports are in place, it will know that mutual support is involved and that sub-assembly or some other trick will be required to place the mutually-supporting blocks simultaneously. It would be possible for the stability test to find all such mutual support cases in advance, but it is easier to ignore them until they are encountered by the planning programs.

Note that moving a single block does not necessitate starting the whole test from scratch. The block that was moved becomes unknown, along with all of the blocks that it was immediately supporting. Any blocks that were involved

in the support of the moved block, all down the support tree, become unknown as well. A number of blocks will thus need re-checking, but often large areas of the scene will not need any checking as a result of the move. Like any other items of secondary data (see section 2.2), the support items can be thrown out when memory gets crowded. Any request for a block's supports will restart the stability test if the items are not found. If-removed methods are present to insure that if any support items are thrown out, a consistent set is thrown out so that no blocks appear to have known supports when some of the relevant data-base entries are actually missing.

Chapter 5: The BUILD Control Structure

5.1 PLANNER and CONNIVER

In the past few years it has become increasingly obvious that the simple recursive control structure of LISP is not really adequate for problem-solving programs such as BUILD. This is not to say that LISP is not computationally general — indeed, both MICRO-PLANNER and CONNIVER are written in LISP — but simply that additional mechanisms are needed for such tasks as recovery from bad choices, creating and invoking demon processes, maintaining multiple world models and multiple processes for exploring them, establishment of a hierarchy of goals and subgoals, and so on.

For years, every programmer working in the area of problem solving had to create his own control mechanisms to handle any of these problems which he encountered. Pseudo-associative pattern matching data bases were frequently re-invented as well. Finally, an attempt was made to create, once and for all, a package of standardized control and data structures that would contain the elements which most problem-solving programs have in common, while retaining enough generality to handle a wide range of problem domains

and levels of sophistication. This package, the PLANNER language, was intended not only to make this type of programming easier, but to eliminate some of the Babel of multiple conventions as well. Full-scale PLANNER is still in the process of being implemented (and, from time to time, redefined) but a subset called MICRO-PLANNER has been available for some time and has, indeed, stimulated a great deal of problem-solving activity.

As increasingly ambitious MICRO-PLANNER programs were attempted, however, problems were encountered with many of MICRO-PLANNER's features. Some of these problems were due to mere implementation details, but others cast doubt on some of the central features of the PLANNER language itself, especially the automatic backtrack control structure. A consensus was reached among many PLANNER users that the language was too specialized; that in far too many situations it provided almost, but not quite, what the programmer needed, and that this, applied automatically, was often worse than nothing at all. In an attempt to provide the user with PLANNER-like capabilities, but with more flexibility and control, Sussman and McDermott created another language, CONNIVER. The PLANNER control structure can easily be implemented in CONNIVER, but alternative control setups are possible as well.

As I mentioned earlier, BUILD was planned with MICRO-PLANNER implementation in mind, but for several reasons, it was ultimately implemented in CONNIVER. The details of the two languages are well documented elsewhere (see <HEWITT 1972>, <SUSSMAN, WINOGRAD, CHARNIAK 1971>, and <McDERMOTT, SUSSMAN 1972>), as is the process that led to CONNIVER's creation <SUSSMAN, McDERMOTT 1972>. It would, therefore, serve little purpose to repeat this information here. Since, however, one of the major goals of my investigation has been to test the capabilities of these two languages in a complex problem domain, I will mention some of the specific problems I had with MICRO-PLANNER and some of the features of CONNIVER which have alleviated these problems.

The first difficulty was encountered in constructing the modeling system. Since MICRO-PLANNER has no mechanism for maintaining distinct multiple world models, it was necessary to tag sets of assertions with a state name. (B1 AT SOMEWHERE) thus became (S B1 AT SOMEWHERE) where S is the name of the state in which this assertion is valid. States could not share common items of data, so each state contained a large number of assertions. Under this system, it was clearly too costly to remember each state in the history of the plan; only one copy was kept, and this was modified at each step. This led to a real problem when, as often happened, BUILD needed to examine some aspect of a

previous state. The old state could, of course, be reached by failing, but only at the cost of losing all information derived more recently. Instead, each block movement was recorded on a list structure (distinct from the MICRO-PLANNER backup tree) and, when necessary, a copy of a previous state could be created from this information. This new state initially contained only block positions; other information was filled in as needed by the various modeling system routines. The data management scheme of Section 2.2 was almost indispensable during this period. None of these problems arise in CONNIVER or in full-scale PLANNER, both of which have multiple data-base contexts and facilities for sharing common data between contexts. In these languages it is entirely feasible to remember each state in the solution of simple problems and several important states in the solution of large problems.

Another major difficulty lies in MICRO-PLANNER's inflexibility with respect to search strategies. Consider the case in which BUILD has made a choice between two alternatives and is well into the exploration of the chosen one when a large but not impossible difficulty is encountered. At this point, BUILD would like to suspend all work on the first path and try the second one, hoping to

find an easier solution. If the second path is just as bad or worse than the first, work on the first path would be resumed from the point at which it was stopped. In MICRO-PLANNER, either goals must be pursued to the bitter end or they must be abandoned entirely, along with all the work invested in them. Note that what is needed is not simultaneous computation, but just the means for saving the state of a process and returning to it later. If the running process can examine the data environments of the suspended processes and use the information found therein, so much the better. This is exactly the sort of multi-process system provided by CONNIVER. Full-scale PLANNER also provides a multi-process capability, though in what appears to be a somewhat clumsier form.

With the choice of a goal or method representing so serious a commitment in MICRO-PLANNER, one would like to be very careful in making such choices, employing heuristics whenever possible to estimate in advance the chances of a path's succeeding and the amount of effort that should be spent before the next choice is tried. Even this is not usually possible, however, since most MICRO-PLANNER choices are made implicitly as a result of multiple matches in the data-base. If, for instance, the system finds several methods under one goal pattern or several assertions

matching a pattern, it will keep the list of these to itself and dispense the members in the arbitrary order in which they were found. Each new element can only be obtained by rejecting the previous one with a failure. Thus, the various possibilities available at the choice point cannot be inspected, pre-sorted, or even counted. It is true that there are ways to defeat this mechanism, but they are painful to use and lead the programmer to wonder whether a language is useful if it must be continually fought with for control of the process. In CONNIVER, instead of trying to handle choices itself, the system makes the list of data-base matches available to the programmer to handle however he wishes, creating at least the potential for well-considered choices. The list of matches is also program-accessable in full-scale PLANNER.

MICRO-PLANNER presented a number of less important but equally irritating problems as well. Failure messages were consistently unable to carry all the useful information back from a failure. Far superior is CONNIVER's ability to save the complete environment of the disaster for post mortems. Since MICRO-PLANNER is not extensive in its range of functions, frequent interaction is required with the LISP system in which CONNIVER is embedded. Due to separate variable bindings and other complications, the two languages

interact awkwardly at best. Occasionally some hapless function would return a value of NIL and a spurious failure would propagate through the system, destroying all data in its path. Properly containing and controlling failures was a constant distraction. In CONNIVER, the interaction with LISP is much cleaner, though still far from optimal, and automatic failure backup does not exist at all. Such things as data-base interrogation functions can be assembled to order from CONNIVER primitives, for instance to implement the mechanisms of Section 2.2. The packaging of PLANNER discourages the use of any but the pre-defined mechanisms.

The reader will have noticed that full-scale PLANNER, as it is currently specified, is not prone to many of the problems that I have outlined for MICRO-PLANNER. In many cases this is because, as problems were pointed out, the specifications for PLANNER were patched to correct them, with each patch and each new feature adding its small increment of size and complexity to an already ponderous system. This is my main objection to PLANNER: While CONNIVER supplies the programmer with a small, elegant set of well-chosen primitives to combine according to his needs, PLANNER tries to anticipate all of the data and control structure needs of its users and provide for them directly. As a result of this difference in philosophy, PLANNER is not

yet available while CONNIVER, its intellectual descendant, has been alive and well for months. In all fairness, however, the theoretical contributions of PLANNER must be acknowledged. Without PLANNER there would be no CONNIVER, and, even if there were, it would be hard to know how to use it without PLANNER's precedent.

5.2 Control Mechanisms Used in BUILD

In the previous section I outlined several objections to the ready-made control structure of MICRO-PLANNER, at least for the specific needs of the BUILD program, and argued that a CONNIVER control structure, specifically constructed for the problem at hand, could avoid most of these problems and lead to much better overall system performance. In this section I will describe the CONNIVER-based control structure that I have devised for BUILD. Despite the emphasis I have placed on made-to-order control systems, I do feel that the BUILD control primitives could, with some modification, be useful in many other types of problem solving. At the very least they will, I think, demonstrate a number of CONNIVER's more interesting features.

In writing this section, I must assume that the reader

has some acquaintance with the CONNIVER language. Since, however, this section is essential to a real understanding of BUILD's operation, I will try to explain a few of the more important points of CONNIVER, in the hope that readers not familiar with this language will be able to comprehend at least the broad outlines of what is to follow. The greatest difference between CONNIVER and LISP is that the control and data environment in which a CONNIVER function is executed can be saved, even after the function has returned or control has left it by means of a GO statement. This is accomplished by creating and saving a tag (or statement label) while the function is running. As long as some external program holds on to this tag, the environment from which the tag came will continue to exist. At any time, the external program can resume execution of the tagged function by GOing to the tag, or it can examine, use, or modify the tagged function's data environment by evaluating other functions relative to the tag. This all adds up to a very flexible sort of multi-process system with excellent communication between the running process and various suspended ones.

In PLANNER, every major function call is accomplished by means of a call to GOAL. The argument to this function is a statement of the desired goal such as (MOVE B1 SOMEWHERE). The data base is searched for one or more

methods which match this pattern, and which thus claim to be able to accomplish this goal. Since, in BUILD, I always want to control exactly which method will be used at any given time, this data base search gains me nothing and is not used. The goal function in BUILD, which, to avoid confusion, I will call BGOAL, performs some housekeeping chores common to all major function calls, then simply evaluates its argument. (BGOAL (MOVE B1 SOMEWHERE)), for instance, simply calls a function named MOVE with B1 and SOMEWHERE as arguments. Selection between various methods is performed either in choosing which goal to call or by code inside the goal functions, not by the random order in which methods are found in the data base.

Most of the housekeeping chores performed by BGOAL consist of binding pointers back into the previous higher-level goal. A variable called REASON always contains a list of the current goal, the goal that called it, and so on up to the top level call to BUILD. By looking at the current REASON, the system can always find out what it is working on and why, in the sense that it knows the higher-level goal of which each goal is a part. Winograd has demonstrated the usefulness of such introspective information in answering external questions, but I include it in BUILD mainly for its

potential usefulness to the program itself. At present this information is only used in trivial ways, but to a more advanced system employing learning and self-criticism some explicit knowledge of its own motives would be indispensable. If the purpose of a goal is to prepare for some other goal, rather than to accomplish something directly, BUILD calls PGOAL instead of BGOAL. These two functions are identical except that PGOAL's REASON is tagged to indicate that it is a preparatory step for some other goal.

BGOAL also saves the old values of CONTEXT and PLAN, so that these critical variables can be restored if necessary. CONTEXT indicates which of the many data-base contexts represents the state of the world at the current point in the plan. PLAN is an ordered list of the steps required to accomplish the top-level goal, and is put together by BUILD as the problem solving progresses. Goals like MOVE, which do something directly rather than by calling other goals, are called primitive goals, and it is these goals which are represented by steps of the plan. Each step includes a statement of the primitive goal, its REASON list, and a pair of contexts indicating the states of the world before and after the step is executed. This PLAN is the output which BUILD produces.

A final function of BGOAL is the invocation of any IF-TRIED methods that match the argument pattern. These are demon procedures, placed in the data base by the user or by other programs, which are activated and run if a goal of some particular form is ever called for. This mechanism is useful for implementing many kinds of temporary program patches. Say, for instance, that we want BUILD to assemble some structure in its usual way, but to avoid touching any objects that are red-hot. This could be accomplished by reprogramming, and this might be the proper course if this restriction is to be permanent, but a temporary restriction could easily be added using an IF-TRIED method for the goal (MOVE ?B ?SOMEWHERE). Whenever MOVE is called, this method would be activated and would check whether B is red-hot. If so, the goal would be aborted. At present, BUILD makes no real use of IF-TRIED methods itself, but this mechanism does look like a useful feature for more advanced programs which could use this sort of temporary self-modification to communicate between higher and lower goals, or to change their own behavior under special circumstances.

Functions analogous to the PLANNER failure backup are performed by what I call a choice and gripe mechanism. Every function which makes a major choice (which block to place next, whether to use sub-assembly or temporary

support, etc.) is called a choice function and must abide by certain conventions: Variables named MESSAGE and BACKTAG are bound but not used in every choice function. Another variable, *CHOICES, has a new tag added to whatever list it already represents. This tag points to a part of the choice function called the gripe handler, whose duties will be explained shortly. Once these conventions are taken care of, the choice function is free to examine its possible choices, pick one that looks good, call any necessary subgoals, and, if successful, return.

Suppose, however, that down in one of these subgoals something goes wrong. In this case a function named GRIPE is called, with an argument indicating the nature of the disaster. If, for example, MOVE were to find B2 in the spot where it wants to place B1, it would pass (HIT B1 B2) to GRIPE. It is GRIPE's duty to report the disaster to the gripe handler of the most recent choice function, which is pointed to by the first tag on the current *CHOICES list. Using this tag for relative evaluation, the MESSAGE variable of the choice function is set to the disaster message which was given to GRIPE, and BACKTAG is set to a tag in the body of the GRIPE function. Control is then transfered to the gripe handler by GOing to the *CHOICES tag.

The gripe handler is in an ideal position to decide

what to do about the problem. The MESSAGE variable gives a terse description of the problem, somewhat equivalent to the PLANNER failure message, but the gripe handler is not limited to such sketchy information as this. Since it is a part of the choice function, it has direct access to any information concerning the choice which it might require. Such things as whether there are other good choices available, why the losing choice seemed best originally, and how good the next best choice appears to be could all be used in its decision. Furthermore, by means of the BACKTAG, the handler has full access to the enviornment of the disaster, and can investigate this for any information it needs. It can even modify this environment, or a copy of it, in order to determine whether the problem can be cleared up by some small change in conditions. If the gripe handler should decide that the disaster in question is not the result of a decision made on its level, but of some higher decision, it simply passes control to the next higher gripe handler indicated on the *CHOICES list.

What remedial actions are available to the gripe handler? First of all, there is the PLANNER solution: Clean up the mess and try a different choice. In BUILD this usually involves restoring the old values of CONTEXT and PLAN, dropping the BACKTAG so that the disaster environment

can be garbage collected, re-arranging some list of possible choices, and jumping back into the choosing part of the choice function. A new choice will then be made, subgoals will be called, and, if all goes well, the choice function will return, just as though this were all happening for the first time. If all the choices on one level are exhausted, the handler can pass control to the next higher handler on the *CHOICES list to try other choices at that level. If this process reaches the end of the *CHOICES list, the gripe message is printed out and CONNIVER goes into a listening loop. This can be viewed either as an admission of defeat on the part of BUILD or as an invitation for the human operator to act as a top-level gripe handler by helping with the solution.

A gripe handler does not always have to abandon the old choice and make a new one. Sometimes, for instance, a choice will be perfectly sound, but trouble will result because of insufficient preparation for a subsequent goal. Consider once again the case where MOVE finds another block where it is trying to put something, and gripes to the choice function which ordered the move. This function's gripe handler can simply call a PGOAL to remove the offending block, and can then try the MOVE again. If the disaster is far removed from the gripe handler in question,

it is possible to reach down into the disaster environment via the BACKTAG, splice in the PGOAL just prior to the call to MOVE, then to restart the MOVE at its beginning, leaving everything between the gripe handler and the MOVE undisturbed. If the system of pointers running through the goal tree is set up properly, this can be somewhat less painful to do than to explain. The same sort of process can be used when the problem requires not preparation, but a slight re-definition of the goal. Often, for instance, the exact position of a temporary support will be arbitrarily chosen, and will later require a slight adjustment to give better support or to avoid an obstacle.

A gripe does not necessarily indicate a fatal disaster; instead, it might indicate that the low level goal is uncertain whether it can or should proceed, and is referring the matter to a process with a more global viewpoint. In such cases the gripe handler can decide to continue with the low level process by GOing to the BACKTAG. A mechanism (currently unused) exists for passing a message back to the lower process, containing clarification, further orders, or perhaps some words of encouragement. If the gripe indicates that the lower process has encountered some unexpected but not insurmountable difficulty, the gripe handler could save the BACKTAG away and try some more attractive possibilities.

If all of these fail, the original goal can be resumed from exactly the point where it left off.

Notice the difference between this control structure and that of PLANNER. Here, each step in the handling of a failure is under the control of an arbitrarily complex user-supplied program with full access to both the environment of the disaster and of the choice which may have caused it. This program can do any of several things, including modification or resumption of the failing goal. Most of the intelligence of BUILD is concentrated at the points where major decisions are made, namely in the gripe handlers and the choice selectors. It is at exactly these points that PLANNER tries to use simple automatic mechanisms. In PLANNER, because of the cost and finality of a failure, it is a good idea to carefully anticipate and correct any difficulties before attempting a goal. Before calling a goal to move a block, for instance, the program should make sure that it is not buried, that the destination is clear, and so on. In BUILD, such anticipation can be used when it results in greater efficiency, but it is equally possible to attempt goals without preparation and to deal with any problems as they are encountered, as interruptions to the main flow of processing. Since a robot would have to deal with a huge number of low probability disasters (lightning,

power blackouts, falling airplanes) this sort of pseudo-interrupt system would seem to offer definite advantages.

Chapter 6: The Planning System

6.1 The Primitive Goals: MOVE and MOVEG

Now that the operation of BUILD's control structure is hopefully clear in the reader's mind, it is possible to describe the detailed operation of the various goal modules that make up BUILD's planning system. The first group of modules to be described are the system primitives, MOVE and MOVEG. These modules do not call any subgoals, but rather check the legality of the step called for and, if possible, add the step to the forming plan. The data base is updated to reflect the action taken. Of course, the special status of these two modules is strictly an artifice of my decision to carry the planning to this level and no farther. In an expanded system, these modules would call subgoals to plan a hand approach, grasp the block or blocks in question, find a path to the destination, and so on, much as is done in Winograd's block system. Ultimately, the system primitives would be specific commands to the mechanical muscles controlling the arm.

The arguments to MOVE are the name of a block to be moved, the AT array of its current location, and the AT array of its destination. MOVE first checks the data base

to determine if the block in question is immovable or if it is already in a place matching some goal state block. In either case MOVE gripes back to its callers. The IN-PLACE gripe is not fatal; if the higher level modules dismiss the complaint, MOVE will continue.

Next, MOVE creates a new working context so that it can make changes in the data base, while preserving the old context to fall back upon in case of failure. The moving block is eliminated from the new context and the resulting scene is tested for stability. Any instability at this point causes an UNSTAB-REM gripe to occur with the exact nature of the instability included in the gripe message. As will be seen later, it is essential for the higher-level modules to know whether an instability problem has occurred with the removal of the moving block or with its placement in the new position; the two cases are handled quite differently.

If the moving block has been removed successfully, MOVE next tries adding it to the scene in the new position. The touch test is run to see whether the block's new position causes it to collide with another object, and, if so, a HIT gripe is generated with a message indicating the objects involved. If there is no collision the stability test is run again, generating an UNSTAB-ADD gripe if trouble is

detected. Again, the gripe message contains a statement of the exact nature of the instability, as determined by the stability test. If MOVE survives all of these tests, it adds to the current plan a new step consisting of the old context, the new context, and the current REASON list with the MOVE call at its head. This plan is returned and the new context replaces the old one.

MOVEG is very much like MOVE, but its purpose is to move a whole group of blocks together as a movable sub-assembly. One block is designated the base, and the present and goal locations given refer to positions of this block. MOVEG is also passed a list of other blocks, called riders, which are supported by the base and ride along with it as the base is grasped by the hand and moved.

The operation of MOVEG parallels that of MOVE very closely. The base and all riders are checked to see whether any are immovable or in place. A new context is created and all of the moving blocks are removed. Stability is tested. The base is then added to the scene in its new position, and all the riders are added in positions which maintain their former locations relative to the base block. This is done by making all of their AT arrays relative to that of the base block, moving that block, then relating the AT arrays back to the fixed three-space of the table. All the moved

objects are then checked for collision and stability, and the plan is altered by addition of the new step. As with MOVE, the appropriate gripe is generated if any of the above tests fail.

MOVEG does perform one function that has no counterpart in MOVE, namely the checking of the movable sub-assembly to see whether it will hold together during transit. This is done by imagining the base and riders to be alone in the universe, with not even the table present and with the base declared immovable. (Now and subsequently, when I say the program "imagines" some situation, I mean that a new context is created in which that situation holds, for the purpose of checking various properties of that state.) The stability test is run and, if any riders fall, the system generates a NOT-MSA gripe with a message indicating which blocks have fallen. To simulate the unsteadiness of the hand, the stability test is run three or four more times, with the gravity vector moved slightly away from the vertical in several directions. The size and direction of these perturbations are programmer-set parameters of the system. Any failure of the riders to withstand the shaking also results in a NO-MSA gripe. Since this testing is fairly time-consuming, the calling module can specify that MOVEG skip this step if the caller is sure that the supplied base

and riders do indeed form a legal movable sub-assembly.

6.2 The Basic Planning Modules

This section will be devoted to the description of BUILD, PLACE, DIGUP, GET-RID-OF, and UNBUILD, five goal modules which together do all of the system's planning except in cases where tricks like sub-assembly and temporary support are required.

BUILD is the top-level goal of the planning system, though it can also be called recursively from some of the other goal modules. Beginning with the current context in which it is called, BUILD assembles a plan to get to the goal state, another context which it receives as an argument. BUILD's first act is to check both the present state (SP) and the goal state (SG) for collision and instability. If problems are found, a BAD-SP or BAD-SG gripe is generated, indicating the exact nature of the difficulty. Thus, BUILD will waste no time trying to achieve an impossible goal state. More valuable than the tests themselves are their side effects: A complete set of touch and support relations for each state is left in the data base. Also included in this pre-testing phase is a check to insure that all of the materials necessary to

duplicate the goal state are present in the current state. Failure to pass this test results in an UNMATCHABLE gripe specifying which SG blocks have no counterparts in SP.

Following the pre-testing phase, BUILD sets SP to a new context sprouted from the old SP, so that any data base changes it makes will be invisible from the outside. Each block in SP is then checked to see whether it is already in a place corresponding to a matching block from SG. It is not necessary that the AT arrays of the two blocks be the same, only that the blocks generate identical models in space; a cube, for example, can match another cube in any of 6 orientations. Whenever one of these pre-positioned blocks is found, an item of the form (IN-PLACE X Y) is added to the data base, where X is the SP block and Y is the SG block it represents. Later, as other blocks are put into place, they too will be marked with IN-PLACE items.

Now BUILD begins its real work. A list is made of all SG blocks that have not yet been placed, as indicated by the IN-PLACE items in the current context. Each of these unplaced blocks depends upon some set of objects for support, as indicated by SUB-BY items in the goal state. The list of unplaced blocks is searched for a block whose indicated supports are all in place. PLACE is then called to place this SG block and, if PLACE is successful, BUILD

starts looking for the next SG block to place. If none of the unplaced blocks has all of its supports in place, a loop of support relations is indicated; BUILD does not currently handle these relatively rare cases, though a slight variant of the scaffolding technique would work in many cases.

Eventually, if no serious trouble is encountered, BUILD will run out of unplaced goal state blocks. Its job is not quite over, however. Recall that in section 1.2 I said that extra blocks in SP, those with no match in SG, could end up anywhere as long as they are "out of the way" of the specified structures. BUILD now must check these remaining SP blocks and, if any are "in the way", call GET-RID-OF to move them away somewhere. Currently, blocks are only "in the way" if they actually touch some block that is in place. Alternatively, a minimum distance could be required, or some portion of the table could be designated as Siberia, with all unwanted blocks going there in the end. This entire phase of BUILD can be cancelled by an optional extra argument from the caller. Once this phase is complete, BUILD is successful and returns the plan it has developed. The actions taken by BUILD upon receiving a gripe from below will be described later.

PLACE receives a block in the goal state as its argument; its job is to find a matching block in the

state, then to call MOVE to put this block into a position corresponding to that of the goal state block. First, a list is created of all the SP blocks with the same shape and size as the SG block to be placed. SP blocks that are immovable or already in place are not considered. If there are several matching SP blocks, the list is sorted according to how deeply each block is buried by other blocks, as indicated by the SUP-BY items in the current context. The block supporting the smallest number of others is passed to MOVE, with the location of the SG block in the goal state being its destination. If the MOVE is successful, a new IN-PLACE item is added to the data base and PLACE returns to its caller.

Several types of gripe can come back to PLACE from the MOVE below it. IMMOVABLE and IN-PLACE gripes should never be received, since these conditions have already been checked for the block in question. HIT and UNSTAB-ADD are complaints that are not really directed to PLACE. These problems are a result of BUILD's choice of an SG block, not PLACE's choice of a matching SP block, so these gripes are passed on up to BUILD. The UNSTAB-REM gripe is handled by PLACE. Since this gripe indicates that the SP block in question cannot be moved without other blocks falling, PLACE calls DIGUP in an attempt to free this block. If DIGUP

succeeds, the MOVE is tried again; if not, PLACE tries the next matching candidate on its list. If PLACE runs out of candidates, it sends a NO-DIGABLE-CANDIDATES gripe on up to BUILD.

BUILD, as we have seen, can receive HIT and UNSTAB-ADD gripes from the MOVE below it, passed along by the PLACE module. The HIT message indicates which block or blocks are in the way of the PLACE, and BUILD calls GET-RID-OF to move each of these blocks to positions where they will cause no further trouble. To avoid wasting the effort already invested by the subordinate PLACE, BUILD reaches down into the environment of the failing MOVE, executes the GET-RID-OF goals just prior to the MOVE, and then re-tries the MOVE, as explained in section 5.2.

The UNSTAB-ADD gripe usually indicates a more serious problem. Since the known supports of the SG block, as indicated by the SUP-BY items, are already in place, an UNSTAB-ADD gripe indicates a mutual support situation, a case where some SG block depends for stability on the blocks it is nominally supporting. This generally means that a trick will be required to place both blocks at once, but often the problem will go away if other blocks are added or removed. For this reason, BUILD saves the gripe message and return tag on a list, keeping hold of the failing MOVE

environment in this way, and loops back to find a different goal state block to give to PLACE. If there are no remaining unplaced SG blocks whose supports are in place, BUILD tries to find unplaced SP blocks which are resting on already-placed blocks and moves these away using GET-RID-OF. If any of these steps is successful, BUILD resumes its normal cycle and eventually will try again to PLACE these SG blocks which failed earlier. Only if it is completely stuck does BUILD pass the list of failure messages on to the routines that try movable sub-assembly and temporary support. If these fail too, a BUILD-GIVES-UP gripe is generated.

GET-RID-OF is a fairly simple module which is passed an SP block, calls FINDSPACE to find a safe location for it, then calls MOVE to put the block there. If FINDSPACE is unable to find room for the given block (a very rare occurrence) a NO-SPACE gripe is generated. If MOVE comes back with an UNSTAB-REM gripe, GET-RID-OF calls DIGUP to free the block. If this succeeds, GET-RID-OF tries the MOVE again; if it fails, the gripe is passed on up to GET-RID-OF'S caller. Unless FINDSPACE has made a blunder, GET-RID-OF should never receive a HIT or an UNSTAB-ADD gripe.

DIGUP is called whenever the attempt to MOVE some block results in an UNSTAB-REM gripe. The SP block in question is

passed to DIGUP, along with a description of the nature of the instability. If this description is not readily available, DIGUP will derive it by imagining the given block to be absent and running the stability test itself. DIGUP extracts from this loss description the names of all the blocks that are immediately unstable and, if all of these blocks are nominally supported by the block being dug up, calls GET-RID-OF against each of them to clear them away. If any of the losing blocks is not supported by the given block, a mutual support condition is indicated. Rather than deal with this problem directly, DIGUP makes a list of all of the blocks that fall when the given block is removed and passes this to UNBUILD, which handles all of the hard cases for DIGUP. Any gripes received by DIGUP are simply passed along to the caller.

UNBUILD receives a list of support-related blocks from its caller, usually DIGUP. UNBUILD'S job is to get all of these blocks safely to the table. This is accomplished by creating a goal state identical to the present state, except that all of the listed blocks are on the table instead of in their current positions. The exact locations of these blocks on the table are determined by FINDSPACE. BUILD is then called to generate a plan for getting from this goal state to the current state. This plan is reversed by

swapping the origin and destination of each MOVE and MOVEG and reversing the order of the steps, and is then grafted onto the plan already formed up to the call to UNBUILD. In this way, all of the construction power of BUILD can be brought to the task of destruction as well.

UNBUILD is one of the system modules that could benefit from some additional development, not because it is failure-prone, but because it tends to produce redundant steps. Often a block will be moved to its FINDSPACE-produced position, and then, almost immediately, it will be moved somewhere else. This could be fixed either by making the goal state generator more intelligent or by waiting until the plan is produced and then cleaning out these redundant steps in a second pass.

Note that throughout the basic system extensive use has been made of the control structure of section 5.2. This structure allows the BUILD-PLACE-MOVE sequence to proceed in a headlong manner, with very little pre-checking of conditions. Trouble is met in a variety of ways when it arises. The system thus works very fast for the usual sort of problem where little or no trouble is encountered. Note, too, that the SUP-BY items in the data base are used in several places to guide the course of the computation. The planning is thus guided in an important way by the system's

very detailed understanding of the problem domain.

6.3 Movable Sub-Assembly

BUILD'S preferred method for dealing with mutual support problems is the technique of sub-assembly, some examples of which were given in section 1.2. When faced with the problem of placing several blocks at once, the system tries to find a movable sub-assembly (hereafter MSA) which contains these blocks, finds a spot on the table to assemble this MSA, calls BUILD to get the assembly done, and finally moves the MSA into place using MOVEG. The modules involved in this planning, besides MOVEG, are TRY-MSA and PLACE-MSA. Except when the sub-assembly technique is specifically disabled by a global system parameter, it is always tried before the temporary support methods are considered. This is because MSA produces shorter plans than the other methods when it is successful, and tends to fail more quickly when it is not.

TRY-MSA is passed the list of UNSTAB-ADD messages and return tags that BUILD has collected in the course of trying to PLACE various objects. The mission of TRY-MSA is to use this information to find an MSA in the goal state which contains at least two currently unplaced blocks, then to

call PLACE-MSA which assembles and moves this MSA. Some of the blocks of the MSA may already be in place but, if only one unplaced block is included, the placement of the MSA will surely fail, since every single-block addition to the structure has already been tried.

TRY-MSA begins by creating a list of possible MSA bases in the goal state. This is done by considering each of the losing states in TRY-MSA'S argument list and finding, for each, what blocks are falling in a pivotal motion. (Recall that the stability test indicates the way in which each block is falling.) The first of these pivoting blocks along a chain of support relations running down from the tentatively placed block (and including this block) becomes a prime suspect. If the support chain branches and leads to two pivoting blocks, no suspect is found. These suspects are current state blocks, so each is replaced by the corresponding goal state block. The suspect list is then filtered to eliminate duplications and to insure that each suspect is involved in the support of at least two unplaced blocks, as determined by the SUP-BY items in the goal state. This procedure usually succeeds in producing a very short list of possible base blocks, typically only one or two in length. In some cases no possible bases are found, and a NC-MSA gripe is sent up to BUILD.

TRY-MSA next picks the first possible base on the list and determines the maximal set of blocks in SG that could ride on this base in a legal MSA. This is accomplished by imagining a state identical to the goal state, except that the table and all other immovable objects have been removed and the base block has been declared immovable. The stability test is then run and any blocks which fall are thrown out of the scene. The scene is then shaken, as described in MOVEG, by slightly perturbing the system gravity vector, and again any falling blocks are eliminated. Whatever blocks remain after this process form the maximal set of riders for the base in question. This process is relatively expensive in terms of computation time, due to the large number of independent runnings of the stability test. This is why so much effort was expended earlier to trim the list of possible bases down to a manageable size instead of, for instance, simply considering all of the SG blocks as bases.

The maximal MSA found by the above process is re-checked to verify that it still contains at least two unplaced blocks, and is then checked to ascertain whether it can be safely put into position. This is done by imagining a state containing all of the SG blocks currently in place, plus the members of the MSA, minus any currently-placed

blocks supported, even in part, by an MSA member. If this structure is unstable, the MSA cannot be placed in the current scene. This is the sort of pre-testing which was skipped in the basic system, but in this case there is the potential for wasting a great deal of work in constructing the MSA, only to find that it cannot be moved successfully into place. If the instability upon placing the MSA takes the form of a pivoting block farther down in the structure, this block is added to the list of possible bases.

At this point TRY-MSA is ready to call PLACE-MSA to get the MSA in question built and moved into place, but first it must decide whether these steps should take place at the current point in the plan or, to eliminate wasted plan steps, at some earlier point. In general, this alteration of earlier plan steps is attempted only if the base block is currently in place and if it was put in place at some point in the current plan, as opposed to being there at the time of the top level call to BUILD. Unless these conditions hold, TRY-MSA simply calls PLACE-MSA and, if successful, returns.

If the PLACE-MSA steps are to occur at an earlier point, the plan is broken into three parts: The first part, containing all of the steps that occur after the placement of the base block, is saved for later consultation. The

second part, consisting of the MOVE step for the base block and any preparatory steps that immediately precede this, is discarded. The third part, containing all those steps occurring prior to the MOVE and its preparation, becomes the current plan and supplies the context at this point to become the current context. Of course, the old plan and context are saved in case the system decides to abandon this exploration. PLACE-MSA can now be called in this reconstructed old environment.

If the PLACE-MSA goal is successful, TRY-MSA could immediately return to BUILD and continue the construction from this point, but instead it tries to make as much use as possible of the subsequent portion of the now-abandoned old plan. Of course, it can not simply assume that this plan fragment is still correct, since the state that it is now working with is different from the state for which the old plan was derived. Each step of this plan fragment must be verified if it is to be used. This is accomplished by trying to execute each of these steps, in order, as primitive goals in the new context, in order to see if any gripes arise. MOVE's whose blocks are not found at the specified origin location are skipped over, since these steps usually represent the placement of blocks which, in the new plan, were placed as part of the MSA. This

verification continues until the remainder of the old plan is exhausted or until one of the steps being verified generates a gripe of any sort. In either case, TRY-MSA simply returns to BUILD, which continues the planning from whatever point was reached. A more advanced system might contain facilities for dealing with gripes encountered during verification and continuing to verify, instead of leaving this mode at the first sign of trouble.

Any gripes received by TRY-MSA from PLACE-MSA cause it to restore the original plan and context, pick the next member of the possible base list, and proceed to find the next possible MSA. If the possible bases are exhausted, a NO-MSA gripe is sent up to the caller. The only exception to this is a HIT gripe received from a subordinate MOVEG step. In this case, TRY-MSA reaches down into the losing context and moves the offending blocks away with GET-RID-OF, just as BUILD does when MOVE generates a HIT gripe.

PLACE-MSA receives a base and a list of riders which its caller, usually TRY-MSA, believes is a legal MSA. PLACE-MSA must find a spot on the table where this MSA can be assembled, imagine the MSA blocks to be in this position, pass this imagined intermediate state to BUILD as a goal, and, if BUILD succeeds in assembling the MSA in the specified place, call MOVEG to move the MSA into the

position it occupies in the goal state. If the least-buried current state match for the base block is resting on the table in the proper orientation, its position is used for the assembly. Otherwise, FINDSPACE is used to pick a position where the MSA blocks will fit. If necessary, this position can conflict with unplaced SP blocks, since BUILD will move these away. Of course, by calling BUILD to assemble the MSA, PLACE-MSA brings the full power of the system to bear on this problem. The recursive seesaw of figure 1-8 is the type of case where this power is useful. TRY-MSA always passes PLACE-MSA a legal MSA, so the MOVEG is instructed to skip the MSA checking phase when it is called from these two modules. Any gripes that PLACE-MSA receives from BUILD or MOVEG are passed up to TRY-MSA.

6.4 Scaffolding and Counterweight

Whenever BUILD finds itself unable to proceed by normal means or by sub-assembly, it tries to find a way of using spare blocks in the scene as temporary scaffolding or counterweights, stabilizing the structure in this way so that more blocks can be added. Examples of both the scaffolding and counterweight techniques were given in section 1.2. The modules involved in this planning are TRY-

TEMP and its subordinates TRY-CWT and TRY-SCAF. Like TRY-MSA, both TRY-CWT and TRY-SCAF can be disabled if desired by the user.

The temporary support modules are far less fully developed than the other portions of the BUILD system. While the sub-assembly technique is basically a single idea that can be applied in a fairly uniform way wherever it is useful, the temporary support category seems to be a loose grouping of a very large number of essentially independent tricks. These tricks can be simple recipes for constructing stable stacks of bricks, complicated recipes for creating bridges (arch or cantilever) over cluttered areas of the table, heuristics for proposing good locations for support structures, methods for dealing with various difficulties that might be encountered, and so on. Ideally, of course, the system should be creative enough to derive from a few basic principles whatever tricks it needs in the given situation and to add these to its repertoire for future use, but this is still somewhat beyond the state of the art. In the meantime, each such trick must be individually programmed into the system. To add to the system every trick that I can think of would thus take years, and would add little to the value of BUILD as a prototypical problem-solving system. Instead, I have tried to program a small

set of tricks which work for a reasonably large class of problems, demonstrate the principles of scaffolding and counterweight, and bring the system to a point where many more features could be added in obvious ways with relatively small increments of programming effort.

TRY-TEMP receives from BUILD the same list of UNSTAB-ADD messages and return tags that was passed to TRY-MSA. Each of these message-tag pairs indicates the unstable result of an attempt to add some unplaced block to the structure being built. TRY-TEMP considers these losing states one by one, and finds the minimal set of additional unplaced SG blocks that must be placed in order for the structure to be stable once again. Such a set of SG blocks is called a stable set or STABSET for this particular loss and is found by imagining various single unplaced blocks to be in place and checking the results. If any of these additions leads to a stable state, the block that was placed is returned. If not, those states in which the original loss was reduced or reversed are noted, and another round of block additions is performed on them. Since the goal state itself is stable, there will always be one such stable set, namely the set of all unplaced SG blocks, though this set may not be minimal. There may in some cases be several distinct stable sets whose placement will counteract a given

loss.

Knowledge of these stable sets is essential for TRY-TEMP's next task, the formation of a list of spare blocks in the current state. If a temporary support structure is built to enable the system to place the losing block under consideration, that support structure must, in general, remain in place until the structure being built is once again able to stand without outside help. The stable set indicates exactly which blocks must be placed to reach this next jumping-off point. Obviously, a block should not be considered a spare block, available for use in a temporary support, if it will be needed as part of the stable set which must be placed. The spare blocks list thus contains all of the SP blocks except the losing block whose placement is being attempted, matches in SP for the stable set members, and SP matches for the SG blocks involved in the support of the losing block and the stable set members. The list of spares is sorted so that the least buried blocks will be considered first. Note that this list may include some blocks that are already in place, as long as these are not an essential part of the structure being worked on.

Once the stable set and list of spare blocks has been determined, TRY-TEMP calls TRY-CWT and TRY-SCAF, in that order. If either of these succeeds, TRY-TEMP returns to

BUILD. Otherwise, it tries a new stable set for the same losing block or, if these are exhausted, goes on to the next loss message on its argument list. If nothing succeeds, it sends a NO-TEMP gripe back to BUILD.

TRY-CWT finds all of the unstable blocks in the losing state under consideration and selects from these a list of all blocks whose centers of gravity are rising, either because these blocks are pivoting or are being pushed up from below. A ray is projected up from each of these centers of gravity and if this ray pierces a level upper surface of the same block, the intersection point is checked to see whether it is covered by other blocks. If this point is clear, TRY-CWT can consider placing a stack of counterweights on it. The necessary weight of this stack can be found by considering the unbalanced force or moment acting on the block in question. Spare blocks are taken from the list one by one until the desired weight is reached and are simply stacked with their centers of gravity in a vertical line. Wedges are laid on their sides in this stack so that their parallel faces can be used.

The counterweight stack thus derived, along with the losing block, the stable set, and the supports of these blocks are passed to BUILD as a goal state. If anything obvious is wrong with this state, such as collision between

counterweight blocks and others, the BUILD will fail at once with an appropriate message. TRY-CWT could read this message and try to move or alter the counterweight tower appropriately, but at present it just goes on to the next rising block or fails back to TRY-TEMP. If the lower BUILD succeeds, TRY-CWT and TRY-TEMP return to the calling BUILD, which dismantles the counterweight stack in the normal course of its planning.

TRY-SCAF is very similar in structure to TRY-CWT. It tries to find uncovered lower vertices of the falling blocks which are moving downward. For each such vertex, it tries to find a tower of bricks and wedges on their sides which will extend from the vertex down to the first level surface beneath it, usually the table. A tower of the proper height is found by considering each of the spare blocks in each of the allowable orientations, and exhaustively trying all combinations of these. The tower may be capped by a wedge lying upon its hypotenuse or by a wedge presenting a slightly slanted upper surface to get a variable height, though in both of these cases the tower must be particularly solid, since it cannot be guaranteed that the applied force will bear directly down on the line formed by the blocks' centers of gravity. Additional modules could of course be added to propose support structures below other points on

downward-moving faces or edges, or to block upward moving vertices from above, as in figure 1-13D. Like TRY-CWT, TRY-SCAF calls BUILD and simply goes on to another vertex if trouble arises.

At present, none of the temporary support modules tries to save steps by going back and modifying earlier portions of the plan. This feature would be essentially the same as that included in the sub-assembly modules, and it was my feeling that one such implementation would be sufficient for demonstration purposes. The temporary support modules are no more likely to fail because of this omission, but they often produce wasted plan steps.

Chapter 7: Concluding Remarks

In this final section I will try briefly to answer a few global questions that arise in relation to BUILD: What guidance can BUILD offer to programmers writing problem-solving systems in different areas? What were the surprises? What loose ends, both major and minor, could use more work? How closely do BUILD'S activities seem to resemble those of humans faced with similar problems? What is the significance of all this? Many of the ideas presented in this section will be quite speculative and non-rigorous but, hopefully, some will be of interest despite this.

Several parts of the BUILD system should be at least potentially useful to problem-solving programmers working on other problems. I think that the methods of storage management described in section 2.2 and the choice-gripe control structure of section 5.2 fall most obviously into this category. While subsequent programs may take radically different approaches from that exemplified by BUILD, the existence of this program will at least give them something very specific to be different from, which can sometimes be as important an organizational factor as a positive metaphor.

Also of possible interest, though somewhat less clear than the features noted above, is the overall style of BUILD. As compared to the typical MICRO-PLANNER program, BUILD seems to have more of a quality which, for want of a better term, I will call looseness. By this I mean a sort of flexibility of behavior, an ability to step back from local jam-ups and look for a way around them, an inability to lock itself into inconsistent states or bad decisions from which there is no escape. The storage management scheme of BUILD contributes to this looseness by making irrelevant the distinction between data that has been derived and data that could be. A whole class of bugs is thus eliminated. The control structure is likewise very loose and flexible. Since the top-level loop of the BUILD module is all-powerful and can work with practically any set of block positions and IN-PLACE items, and since the system gets back to this loop after practically every step, it is very hard for BUILD to get itself into some kind of mess from which it cannot proceed. People, of course, possess extreme degrees of looseness, including even the ability to alter their own procedures.

There were two major surprises for me in the course of programming BUILD. The first of thse was the extreme ease of programming the planning system. The second was the

extreme difficulty of programming the modeling system, especially the stability test. About 80% of the programming effort went into the modeling portions of BUILD, representing something on the order of a year of effort on my part. I would claim, moreover, that this split owes very little to my own particular tastes and programming style. My initial estimate was that the division of programming effort would be more like 60% planning system to 40% modeling, and I made several attempts to replace modeling power with planning heuristics, all to no avail.

The reasons for this disparity of difficulty are not hard to find, at least after the fact. First, the linearization of a basically simultaneous, multiply entangled equilibrium problem like block stability, so that what is going on can be understood by another program, is simply not an easy problem. (In a moment I will explain my theory of why three-year-old humans can do it.) On the other hand, once this problem is solved and good descriptive data is available to the planning system, it is almost always clear what to do at each point in the planning. No doubt some readers were disappointed at the lack of flashy heuristics in the planning system, but the reason for this should now be clear: Heuristics are simply not needed in a system that has a good understanding of its problem. The

ease of programming the planning modules was also greatly aided by the relative ease of using CONNIVER, as compared to MICRO-PLANNER, and by the way that the control structure broke the problem into relatively independent sets of choice selectors and handlers for specific gripes.

A number of minor extensions to BUILD have been suggested in the earlier chapters. Any number of small changes could be made to the planning modules to handle assorted cases where the system now fails or is inefficient, though actually making these changes would probably only be useful if someone really wanted to use the system for practical purposes or needed practice in programming. Somewhat more useful would be a real hidden line eliminator for the display system.

One example of a non-trivial extension would be to extend the system downward to the level of actual joint movements of a modeled mechanical hand. Not only would this involve changes to the modeling system and a whole new set of goal modules to write, but it would force the programmer to develop a much better way to model empty space than is currently used in BUILD. The blind search and test algorithm used by the current FINDSPACE is just barely adequate for locating spaces on a nearly empty two-dimensional table; it would be hopelessly lost in trying to

find a path through 3-space for a hand carrying several blocks.

What the system really needs is a good way of dealing with the very peculiarly shaped non-object that is the empty space in the scene, of finding the major blobs that comprise it and the size and shape of the connections between these blobs. One obvious way of getting at these relationships would be to employ a mapping scheme, dividing the 2-D table surface or 3-D space up into squares or cubes and marking each of these according to whether it is occupied or not. The visual image of the table seems to function as a convenient map for humans; any area of the table can be quickly checked for occupancy, and the large blobs of empty space can be easily picked out. It might also be noteworthy that people, except possibly for the blind, do not seem to be very good at doing 3-D FINDSPACE tasks without reference to visual inputs; when they want to find whether doors will open or switches can be reached in a crowded area like a space capsule, they usually build a model or make a sketch.

The BUILD control structure seems, for the moment, to need little added to it, since some of its powers are not even used by the present system. It could, of course, use some polishing to make it easier to use and more attuned to general needs than to the specific needs of BUILD. Two

control structure areas do, however, come to mind as candidates for significant future work. These areas are the detection and avoidance of infinite loops, and the area of inter-goal co-operation.

The very looseness of BUILD'S control structure which keeps it out of jam-up trouble seems to make it prone to various types of infinite loops. In the present system this has been dealt with on a case-by-case basis, but I feel that a general solution to the looping problem should be a part of more advanced systems. It is notable, in this regard, how free humans seem to be from looping, without any conscious effort to avoid it. What seems to be happening is that a fairly complete trace or record of the person's mental activities is saved, at least for a short time, and some system demon occasionally checks the current mental state against various previous states. If a match is found, the demon sends an interrupt to some higher level which could either repair the looping process to make it converge, or drop it and go on to something else. Whether or not this is an accurate view of human functioning, such a module could be of great benefit to problem-solving programs, eliminating another whole class of bugs. The format of the record kept, the way this record is indexed or hash-coded, and the nature of the matcher would, of course, require

careful thought.

By inter-goal co-operation, I mean the protection of the accomplishments of one goal from premature destruction by later goals, and the dissemination of useful information found by one goal to all of the other goal modules that might be interested. BUILD seems to be a very bad program for studying these things, since block building requires very little of this type of co-operation. The only results that need to be protected are block locations, handled by the IN-PLACE items, and possibly empty spaces that have been cleared. For a while, I was protecting these empty spaces by putting "ghost" blocks in them, but this procedure proved to be more trouble than it was worth, so now empty spaces are not protected. The system, after all, can always find another one. The only inter-goal communication used at present is between superior and inferior goals by means of arguments and gripes, though the mechanism of IF-TRIED methods (see section 5.2) could be used for some other types of communication. I suspect, however, that all of this could be put into some reasonably clean and elegant form if some work were done on this problem.

In commenting about whether BUILD seems to work on its problems in ways at all similar to the approaches taken by people, I should emphasize at the outset that my

speculations are based primarily on rather haphazard introspection, not on careful psychological testing. Nevertheless, some interesting similarities and differences seem to stand out. The control structure, while far from human in its capabilities, does seem better able to model human behavior than, say, the predicate calculus or Newell and Simon's production systems <NEWELL, SIMON 1972>. Such human activities as setting up explicit goals, testing hypotheses, switching back and forth between two equally promising branches, and giving up with some specific complaint all seem to have clear counterparts in BUILD, while they cause trouble for other theories. The most striking difference between BUILD and humans, aside from BUILD'S inability to learn, seems to be that BUILD'S models are quantitative and depend heavily on floating-point arithmetic, while people seem to prefer qualitative descriptions. While BUILD has to work to discover that two blocks are touching along a face or aligned at an edge, a person is likely to define their relative positions in terms of such facts and to have to work to determine that some block's center is 6.3 inches from the table and 2.8 inches from the back wall.

BUILD'S approach to stability testing also seems to be quite far from the usual human approach. Since BUILD has no

access to the real world, it cannot propose steps that it thinks might work and then test them by actually trying them. Since BUILD must be its own harshest critic, its stability test cannot settle for rules of thumb, hypotheses, and guesses. In this respect, BUILD'S problem solving is closer to that of an engineer designing a bridge than that of a child playing with toys. BUILD must have a stability tester that can deal correctly with the worst cases it will ever encounter, so it uses this powerful module on the simple cases as well.

A child, on the other hand, is likely to develop a very broad and shallow stability tester that is made up of a number of modules. At first, he might only know a few simple rules: A brick resting on the table will not fall; a brick resting on another brick will not fall unless it hangs over too far; a brick on a slant will slide down until it hits something. Later, he might figure out or be taught the concept of center of gravity, at least for bricks. Using this, he can work problems by visualizing spatial motions: If the block tips over this way, it will have to lift the center of this heavy block; therefore, it won't. Finally, in high school or college, he will be taught about forces and will be able to deal with lever arms and friction. Incidentally, adding some of these special rules for easily

recognized special cases (block flat on table, non-overhanging stacks, etc.) could vastly speed up BUILD's current stability test. In retrospect, I probably should have tried adding this conglomerate type of test to BUILD instead of working on the single powerful test, though I would probably never have been able to deal with complicated things like friction.

Finally, we come to the difficult question of BUILD'S overall significance. It is my feeling that BUILD'S major contribution to the field of artificial intelligence is in bringing actual practice up to the level of theory. The arrival of PLANNER, Winograd's blocks program and, later, CONNIVER led most AI researchers to believe that broad new areas of problem solving had suddenly been made far more accessible. Many, if asked to think about the problems attacked by BUILD, could have sketched out the outlines of such a system with little trouble. But having a vague idea of how to do something is a very different thing from having a working program. My own original conception of BUILD'S overall structure bears surprisingly little resemblance to the program of today, though this can partly be blamed upon the mid-course change of language. With BUILD as a solid base to stand upon, we can look farther into the uncharted territory and begin to discern the vague outlines of the

next generation of problem solvers.

BIBLIOGRAPHY

<PLUM, GRIFFITH, NEUMANN 1970>
Plum M, Griffith A K, and Neumann B
A Stability Test for Configurations of Blocks
AI MEMO 188
M.I.T. Artificial Intelligence Laboratory
February 1970

<BOBERG 1972>
Boberg, Richard
Generating Line Drawings from Abstract Scene Descriptions
Master's Thesis
M.I.T. Department of Electrical Engineering
October 1972

<HEWITT 1972>
Hewitt, Carl
PLANNER
AI-TR-258
M.I.T. Artificial Intelligence Laboratory
April 1972

<McDERMOTT, SUSSMAN 1972>
McDermott D V and Sussman G J
CONNIVER Reference Manual
AI MEMO 259
M.I.T. Artificial Intelligence Laboratory
May 1972

<NEWELL, SIMON 1972>
Newell A and Simon H A
Human Problem Solving
Prentice-Hall
Englewood Cliffs, N.J. 1972

<PAUL, FALK, FELDMAN 1969>
Paul R, Falk G, and Feldman J A
The Computer Representation of Simply Described Scenes
Stanford AI MEMO 101
Stanford Artificial Intelligence Project
October 1969

<SUSSMAN 1971>
Sussman, Gerald Jay
The FINDSPACE Problem
Vision Flash 18

M.I.T. Artificial Intelligence Laboratory
August 1971

<SUSSMAN 1972>
Sussman, Gerald Jay
Teaching of Procedures - Progress Report
AI MEMO 270
M.I.T. Artificial Intelligence Laboratory
October 1972

<SUSSMAN, McDERMOTT 1972>
Sussman G J and McDermott D V
Form PLANNER to CONNIVER - A Genetic Approach
Pages 1171-1179
Proceedings of the Fall Joint Computer Conference, 1972
AFIPS Press
1972

<SUSSMAN, WINOGRAD, CHARNIAK 1971>
Sussman G J, Winograd T, and Charniak E
MICRO-PLANNER Reference Manual
AI MEMO 259
M.I.T. Artificial Intelligence Laboratory
May 1972

<WINOGRAD 1972>
Winograd, Terry
Understanding Natural Language
Academic Press
New York 1972

<WINSTON 1970>
Winston, Patrick H
Learning Structural Descriptions from Examples
AI-TR-231
M.I.T. Artificial Intelligence Laboratory
September 1970

<ZUKHOVITSKY, AVDEYEVA 1966>
Zukhovitsky S I and Avdeyeva L I
Linear and Convex Programming
Saunders
Philadelphia 1966

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 4/30/96

Report # AI-TR-283

Each of the following should be identified by a checkmark:
Originating Department:

- [x] Artificial Intellegence Laboratory (AI)
- [ ] Laboratory for Computer Science (LCS)

Document Type:

- [x] Technical Report (TR)
- [ ] Technical Memo (TM)
- [ ] Other:_____

Document Information

Number of pages: 143 (149-IMAGED)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:

- [x] Single-sided or
- [ ] Double-sided

Intended to be printed as :

- [ ] Single-sided or
- [x] Double-sided

Print type:

- [ ] Typewriter
- [ ] Offset Press
- [ ] Laser Print
- [ ] InkJet Printer
- [ ] Unknown
- [x] Other: copy

Check each if included with document:

- [x] DOD Form
- [ ] Funding Agent Form
- [x] Cover Page
- [ ] Spine
- [ ] Printers Notes
- [ ] Photo negatives
- [ ] Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

	Description :	Page Number:
(A)	IMAGE MAP!	(1-143) UNNUMBERED TITLE PAGE, UNNUMBERED ABST, UNNUMBERED ACKN, 4-11, UNNUMBERED FIG 13, UNNUMBERED FIG'S (3), 17-24, UNNUMBERED FIG, 26-34, UNNUMBERED FIG'S (2), 37-38, UNNUMBERED FIG, 40-143 (144-149) SCANCONTROL, COVER, DOD, TRGT'S (3)
(B)	MARK IN RIGHT MARGIN FROM XEROXING	

Scanning Agent Signoff:

Date Received: 4/30/96 Date Scanned: 5/6/96 Date Returned: 5/9/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Artificial Intelligence Laboratory

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP
None

3. REPORT TITLE
A Planning System for Robot Construction Tasks

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Thesis -partial fulfillment of requirements for B.S. & M. S. in Electrical Engineering March 1973

5. AUTHOR(S) *(Last name, first name, initial)*
Scott E. Fahlman

6. REPORT DATE
May 1973

7a. TOTAL NO. OF PAGES
143

7b. NO. OF REFS
13

8a. CONTRACT OR GRANT NO.
N00014-70-A-0362-0005

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
AI TR-283

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES
None

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency
3D-200 Pentagon
Washington, D.C. 20301

13. ABSTRACT

This paper describes a system which plans the construction of specified structures out of simple objects such as toy blocks. The planning is done using a 3-D model of the work space. A powerful control structure allows the use of such techniques as sub-assembly, temporary scaffolding, and counterweights in the construction.

14. KEY WORDS

DD FORM 1 NOV 68 1473 (M.I.T.)

AI-TR283 Cover ii

UNCLASSIFIED

Security Classification

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**