"AND TREE" COMPUTER DATA STRUCTURES FOR SUPERVISORY CONTROL OF MANIPULATION

by

PHILIP A. HARDIN
S.B., Massachusetts Institute of Technology
(1965)
M.S., Massachusetts Institute of Technology
(1966)
MECH.E., Massachusetts Institute of Technology
(1969)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF
PHILOSOPHY
at the
MASSACHUSETTS INSTITUTE OF
TECHNOLOGY

October, 1970 (i.e. Feb 1971)

Signature of Author... Signature redacted ...
Department of Mechanical Engineering, October 13, 1970

Certified by... Signature redacted ...
Thesis Supervisor

Accepted by... Signature redacted ...
Chairman, Departmental Committee
on Graduate Students

Archives

"AND TREE" COMPUTER DATA STRUCTURES FOR SUPERVISORY CONTROL OF MANIPULATION

by

PHILIP A. HARDIN
S.B., Massachusetts Institute of Technology
(1965)
M.S., Massachusetts Institute of Technology
(1966)
MECH.E., Massachusetts Institute of Technology
(1969)

SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF

PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF

TECHNOLOGY

October, 1970 (i.e. Feb 1971)

Signature of Author..
Department of Mechanical Engineering, October 13, 1970

Certified by..
Thesis Supervisor

Accepted by..
Chairman, Departmental Committee on Graduate Students

Thesis M.E. 1971 Ph.D.
Archives

"AND TREE" Computer Data Structures for Supervisory Control of Manipulation

by

Philip A. Hardin

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements of the degree of Doctor of Philosophy, October, 1970.

ABSTRACT

Remote manipulators, originally designed for handling radioactive materials, have been proposed for use on unmanned space vehicles. A pure time delay in the transmission of radio signals, present because of distance or other reasons, makes remote manipulation task completion more difficult. A supervisory controlled computer-manipulator overcomes this problem and has many other potential uses.

This thesis describes a supervisory controlled manipulation system which can accomplish complex manipulation tasks. The system, which presumably knows the initial positions of all objects involved in the task (the initial state of the task), is given an operator's description of the desired final goal state of the objects in the task. It uses this description to generate a set of sub-tasks, each of which describes moving one object to a final position designated by the goal state description.

The system is divided into two parts. The upper level is an AND TREE which orders the sub-tasks so their combined solution results in the solution of the complete task. The lower level consists of

1. a procedure to generate the state spaces which describe a sub-task, and

2. a set of shortest path algorithms which find a path through the state spaces and therefore find the solution of the sub-task.

The system was implemented on a digital computer, and the manipulator jaws and object of the task site were simulated by a two dimensional computer model. Six example tasks, solved by the system are presented: two in step by step detail. In addition, there are included: detailed descriptions of task types for which the system will find solutions (if they exist!); the additional abilities the system would have to possess to solve more complex tasks; the economic advantages of this system as compared to others that have been proposed; and documentation of the system as implemented.

Thesis Supervisor: Thomas B. Sheridan
Title: Professor of Mechanical Engineering

ACKNOWLEDGEMENTS

I deeply appreciate the advice, guidance, and encouragement provided by my thesis committee: Professor Thomas B. Sheridan especially for his patience and support; Professor Joseph Weizenbaum especially for his helpful ideas; and Professor Daniel E. Whitney especially for his constructive advice.

I also thank Professor Marvin L. Minsky and other members of Project MAC's Artificial Intelligence Group for their assistance in using their PDP-10 computer; Jane Browning for the excellent typing; and my wife, Jane, for editing, drawing the figures, and preliminary typing.

The work of this thesis was supported by the National Aeronautics and Space Administration Grant NGL-22-009-002.

TABLE OF CONTENTS

ABSTRACT 2

ACKNOWLEDGEMENTS 4

CHAPTER I. INTRODUCTION 8

Background 8

Requirements of a Supervisory Controlled Manipulator 22

Similar Work 23

Discussion of the Problem 25

Preview of the System Proposed in This Thesis 27

Preview of the Remainder of This Thesis . . 37

CHAPTER II. BASIS FOR THIS SYSTEM 39

Characteristics of an AND TREE 39

The TASK TREE 50

CHAPTER III. SHORTEST PATH ALGORITHMS 54

Flooding Algorithm 55

A* Algorithm 81

Two Stack Diamond 83

CHAPTER IV. IMPLEMENTATION OF ALGORITHMS FOR THE PURPOSES OF THIS WORK 87

Restrictions on the Examples and Demonstrations and Definitions 87

Some Basic Manipulation Situations 95

Generation of a State Space for an Object of Any Shape 101

Moving an Object from its Initial Position to a Final Position 107

Discovering Which Objects are in the Way . 111

Finding the Order in Which to Move Several Objects to Final Positions. . 113

CHAPTER V. CAPABILITY OF THIS SYSTEM TO FIND SOLUTIONS. 117

Tasks for Which This System is Not Assured of Finding Solutions. 117

Tasks for Which This System Will Find Solutions 123

CHAPTER VI. SELF-DIAGNOSIS OF FAILURE. 129

CHAPTER VII. ECONOMIC ADVANTAGES OF THE SYSTEM. 132

CHAPTER VIII. EXAMPLES OF TASKS SOLVED BY THE DEMONSTRATION SYSTEM 137

Two Examples Presented in Detail 138

Four Examples Briefly Presented. 162

CHAPTER IX. SPECIFIC PROBLEMS FOR FUTURE WORK. 177

What is a "Handle"?. 177

Variable Quantization of a Space 182

Not Enough Out of the Way Places 184

Pushing or Carrying More Than

One Object 187

Finding Connected Out of the Way Places. . . .190

Use of an N Level Method to Find

Problem Solutions. 190

CHAPTER X. CONCLUSIONS. 191

APPENDICES 193

REFERENCES 225

BIOGRAPHY 228

Chapter I Introduction

Background

Manipulators allow man to extend the range of the environment he can affect into those that are very distant or very dangerous, without the hardship or hazard that would be attendant if he were physically present.

The first remote manipulators (master-slave manipulators) were built after World War II at Argonne National Laboratory when ways were needed to conduct experiments with radioactive materials.[9*] These manipulators did not increase man's strength or precision, but skilled operators could perform delicate and complex tasks with them. These early manipulators had mechanical linkages with steel tapes and pulleys; the recent manipulators have been built using servo motors and electrical linkages.

There has been much improvement in these manipulator systems, now called Teleoperator Systems, since the first ones were built. They have become quite intricate, and proposals have been made for systems that include stereo T.V. (for eyes) and full duplication of the operator's arm motions.[14, 15] Proposals for their use have extended from

* Superscripts indicate references

radioactive materials experiments, to use in outer space, to aids for the physically handicapped.

These systems have several drawbacks when their use is extended to fields for which they were not specifically designed. For example, they only duplicate an operator's motions. For repetitive tasks this constraint may limit their usefulness.

As another example, consider the slave hand being a long distance from the master controller. In this situation there is a pure transmission delay of the electrical signal. For example, if the master end is on Earth, and the slave is on the moon, any motion of the master end will require nearly three seconds to be confirmed because of the distance the signals must travel. If systems are to cope with emergencies that require action in less than round trip signal time, they must have some method to handle emergencies locally.

Ferrell[7] has shown that manipulation tasks executed by a man using a master-slave manipulator with no force feedback can be completed in the presence of a pure time delay, and that the time to complete the task is linearly proportional to the delay. A manipulation task can be accomplished in a reasonable length of time with a five or ten second time delay (Ferrell used delays of this order), but attempting any

but the simplest tasks with a five minute delay would require very long completion times. Delays of five minutes or more are typical for communications with the near planets.

To overcome the difficulty of operating under conditions of long delay times, manipulators which have intelligence have been proposed by Sheridan,[27,8] Johnsen,[13] and others. The machine proposed by Sheridan would be operated in a supervisory manner; the operator would give it instructions and would periodically check on its progress.

Several researchers have worked on systems that are supervisory controlled. McCandlish[16] found, in the case of his experiment, that his subjects performed worse on a given simple task when using a supervisory controlled system than when they were in continuous control of the task. In later interviews the subjects disclosed that they had fairly well memorized the entire task, and could perform it reasonably well in an open-loop mode. Their impressions were that they worked much harder on the task when continuously controlling it; they "relaxed" and made unnecessary errors when operating in supervisory mode. Also, they were curious enough about the system's performance when operating in supervisory mode to increase the value of their task penalty function by requesting extra feedback to verify the system's performance.

Barber[2] has also investigated the use of supervisory controlled manipulation systems. He devised a compiler system to enable an operator to symbolically input commands to a computer controlled manipulator. An example of a command is "Move left 200 units or until touch [something]." The system also has the facility to receive several commands as a set, accept a set name, and execute the entire set. Barber gives as an example a command set that would cause the manipulator to search a plane area (using its touch sensors), and if it finds a block, to move it to a specified position in the plane.

Unfortunately, Barber reported that he had many difficulties with the hardware in his system, which prevented him from doing any more than testing the basic aspects of his system. Also, it was found that to have a flexible and easily utilized software system, the computer program would have to have major changes made in it. Barber did outline the improvements that were needed, but they were never implemented and no extensive experiments were performed with the equipment.

Barber's ideal system overcomes most of the problems of manipulation with long time delays. The operator can specify the manipulator motions before their use, and, with

artful use of the conditional statements, avoid or prepare for emergencies or slight errors in the on-site manipulator's performance.

But Barber's system does have one drawback. The operator must explicitly, and in detail, specify all actions the manipulator is to make. It is true he must do this only once, but specifying all the motions and conditional actions of even a relatively simple task can be a laborious undertaking. It would be better for the operator to specify _what_ he wanted done rather than _how_ to do it.

Whitney[31] developed a supervisory controlled system for manipulation that allows the operator to specify only _what_ is to be done; the computer, using optimal control techniques, figures out _how_ to perform the task.

Whitney's approach is to define a set of atomic manipulator primitives: for example, move left one unit, move up one unit, open, grasp, etc.; and to digitize the physical space in which the task is to be performed. The digitized model of the space (that includes the objects) and the manipulator primitives are combined to form a _state space_ that describes the task.

The _state_ of a physical space is determined by the location, temperature, or other variables of interest that

describe the space or the objects within it. A specific *state*, then, corresponds to a particular condition of the physical space.

The term state space is used in the same sense as in control theory.[30] Specifically, a *state space* is a space in which there is one dimension for each degree of freedom (that is, for each variable of interest), of the physical space. In manipulation tasks, typical degrees of freedom are the variables which describe the manipulator jaw coordinates and the coordinates of the object or objects to be moved. In a state space, each variable is allowed to assume values over the range of interest. The volume of the state space is the product of the range of the allowed values of all variables. For example, if in a digitized space, the variables of interest are A, B, C, ... , Q; and A is allowed to assume "a" discrete values, B is allowed to assume "b" values, etc., then there will be a•b•c• ... •q points in the state space.

The *goal* of a manipulation task is defined as a desired state of the task space. The solution of a manipulation task is a *path* through the state space from the current state to the desired state.

State spaces are designed so that the difference between neighboring states is a single, simple feature. Neighboring states of a state space describing a manipulation task differ by a feature alterable by a single atomic manipulator primitive; one can move from state to state and alter the environment by the application of a string of manipulator primitives. Hence, a path from a current state to a goal state is an ordered set of manipulator primitives which, when executed in the physical space, will accomplish the desired task. Whitney's method reduces the entire problem of finding solutions to manipulation tasks to generating a state space and finding a path through the state space.

As an aid in understanding the concepts presented above, consider the following example, summarized from Whitney.[31] Figure 1 shows a one dimensional line with a set of manipulator jaws and a block. The jaws can move along the line and open and close. If it is assumed that the object will not be moved, the state vector necessary to describe changes in the space is

$$S = \begin{bmatrix} X_j \\ H \end{bmatrix}$$

where X_j=X coordinate of jaws = 1, ..., 5

$$\text{and } H = \begin{cases} 0 \text{ if the jaws are open} \\ 1 \text{ if the jaws are closed.} \end{cases}$$

Physical space from Whitney[31] $S = \begin{bmatrix} X_j \\ H \end{bmatrix}$ = state vector

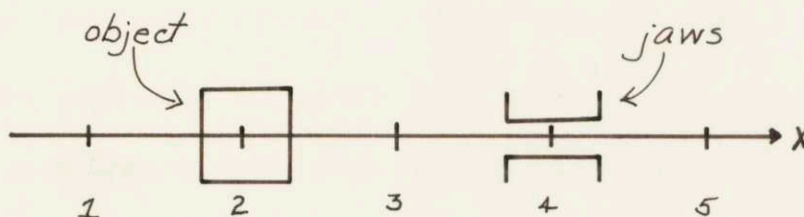
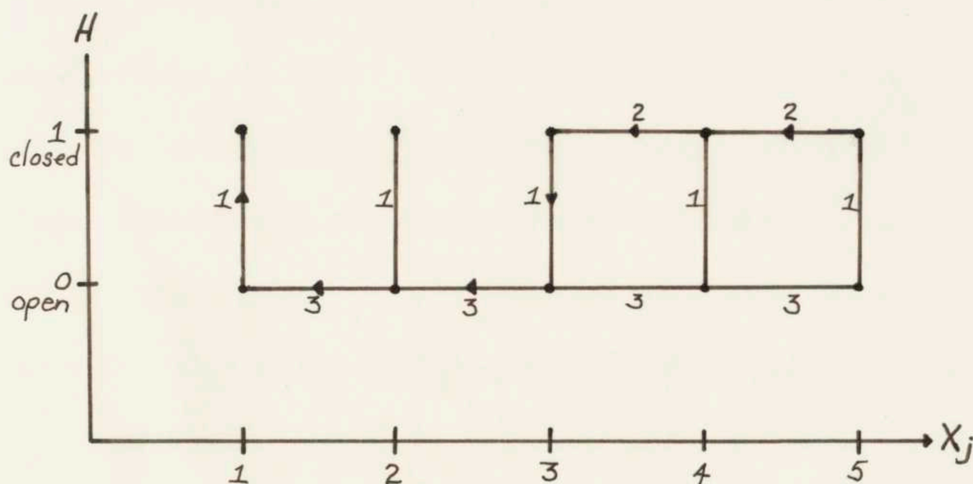


Figure 1



State space and allowable transition of Figure 1, from Whitney.[31]

Nodes represent states, edges represent possible transitions.

Numbers along edges represent cost of transition.

Arrows show path from $S = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$ to $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$

Figure 2

The relevant atomic commands for this task are open jaws, close jaws, move jaws one unit right, and move jaws one unit left. The state space for figure 1, with allowable transitions, is shown in figure 2. The possible states are indicated by the nodes (points). The edges (lines) connecting the nodes indicate the possible transitions from state to state. The jaws cannot move while closed from $X_j=1$ or $X_j=3$ to $X_j=2$ because they will collide with the object. The numbers along the transition lines represent the cost of making the transition. The charge is one unit for opening or closing the jaws, two units for moving with the jaws closed, and three units for moving with the jaws open.

Now let us investigate solving a simple task, moving the jaws from their present position, state $\begin{bmatrix}5\\1\end{bmatrix}$, to X=1, closed, or to state $\begin{bmatrix}1\\1\end{bmatrix}$. The shortest path algorithm finds the path to the goal as the sequence of states (in the form $S_i = \begin{bmatrix}X_j\\H\end{bmatrix}$):

$$S_1 = \begin{bmatrix}5\\1\end{bmatrix},\ S_2 = \begin{bmatrix}4\\1\end{bmatrix},\ S_3 = \begin{bmatrix}3\\1\end{bmatrix},\ S_4 = \begin{bmatrix}3\\0\end{bmatrix},$$

$$S_5 = \begin{bmatrix}2\\0\end{bmatrix},\ S_6 = \begin{bmatrix}1\\0\end{bmatrix},\ \text{and } S_7 = \begin{bmatrix}1\\1\end{bmatrix}.$$

Translating, the jaws are moved from X=5 to X=3 closed, opened at X=3, moved to X=1 open, and then closed, which

solves the given task. The arrows on the transition lines in figure 2 show the path in the state space.

To plan tasks which move the block, the state vector must be extended to include the block's X coordinate,

$$S=\begin{bmatrix} X_j \\ H \\ X_o \end{bmatrix} \qquad \begin{matrix} X_j \text{ and H as before} \\ X_o\text{=X coordinate of object} = 1,\ldots,5 \end{matrix}$$

and the open and close motions of the jaws at X=2 must be interpreted as release and grasp, respectively. By extending the state space to three dimensions, any task to move the object can be solved using the principles we have discussed. The complete task is thoroughly examined by Whitney.

To summarize, Whitney showed that the solution of a manipulation task can be found as a path through a state space. This path describes a string of manipulator primitives which, when executed in sequence, results in accomplishing the requested task.

As a demonstration of his theory, Whitney implemented a version of the state space system on a PDP-8 computer[5] that controlled a three degree of freedom manipulator (X,Y, open-close jaws). The manipulator system moved one

unit square blocks in response to task requests input from a teletype.

The state space method provides a solution technique for manipulation problems. But it has a realistic limit on the complexity of tasks it can solve. Because the state space description must be contained in a computer or some storage medium accessible by a computer, the size (number of nodes) of the state space is limited. From experience, this limit on task complexity has been set so that the state space describes moving one object.

In an attempt to overcome this limitation, Whitney proposed a method that could find the solution to a task that required moving two or more objects. The operator specifies all the relevant sub-tasks (sub-task means a task in which one object will be moved) of which he expects one permutation to be optimum, and the system forms an OR TREE of all possible permutations. It evaluates all these combinations to find the cheapest chain of sub-tasks that accomplishes the task. The system finds the cost and solution of each sub-task using the the state space method.

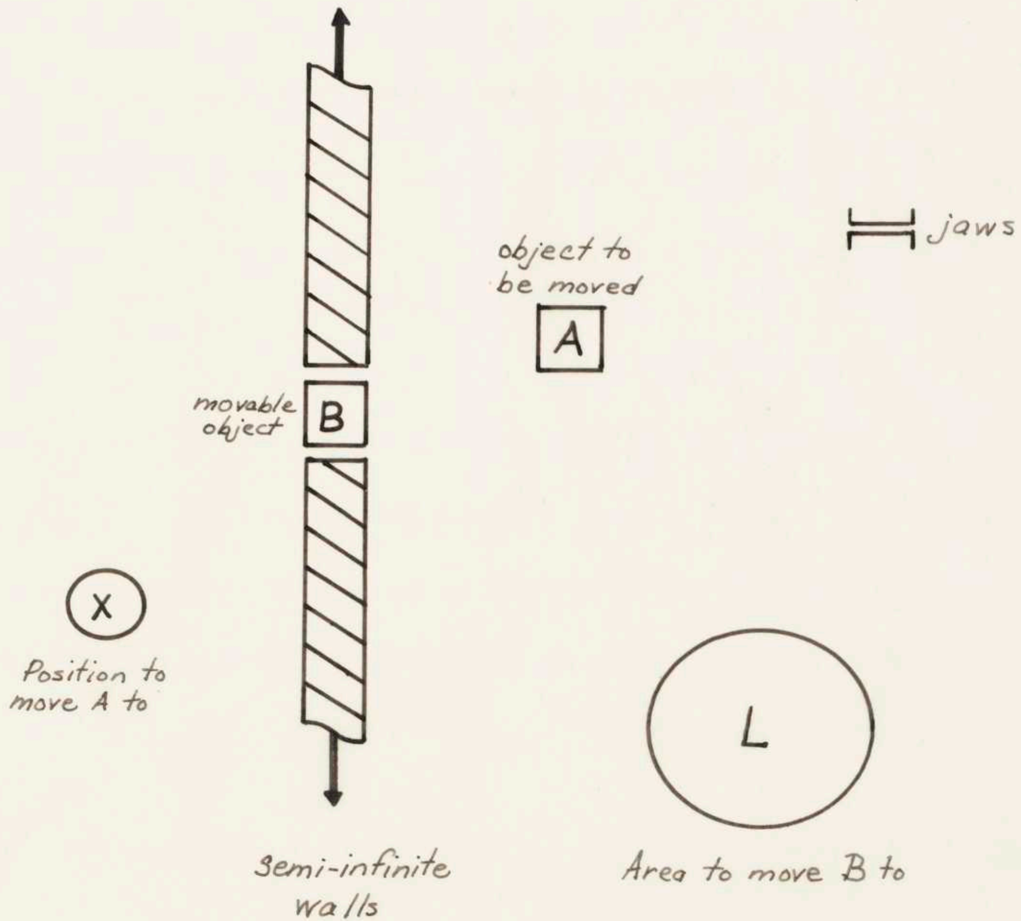
One disadvantage of this method is that the operator must specify all sub-tasks the system is to consider. In many cases, this includes sub-tasks in which the operator

has no interest, except that they must be solved before the complete task is solved. In these cases, the operator is helping the system find the solution by specifying, in part, how the task is to be solved.

As an example, consider the task shown in figure 3. The operator is interested in moving A to X. But, in addition, he must specify that B should be moved, and where it is to be moved to, in this case, the area L. Let us say the area L contains three distinct positions, L_1, L_2, and L_3. The OR TREE which the system computes is shown in figure 4. The costs are all different as the jaws move B to different places, then return to move A. The system has to compute seven paths, and remember and compare their costs. Of course, a 7-dimensional state space could solve it automatically, without the operator giving any hints at all, but seven dimensions are out of proportion for such a simple problem.

In summary, the OR TREE of concatenated paths is one system to move several objects to several places, but it has the disadvantages of

1) in some cases the operator must specify, in part, how the task is to be solved by specifying all the sub-tasks that need to be solved for the



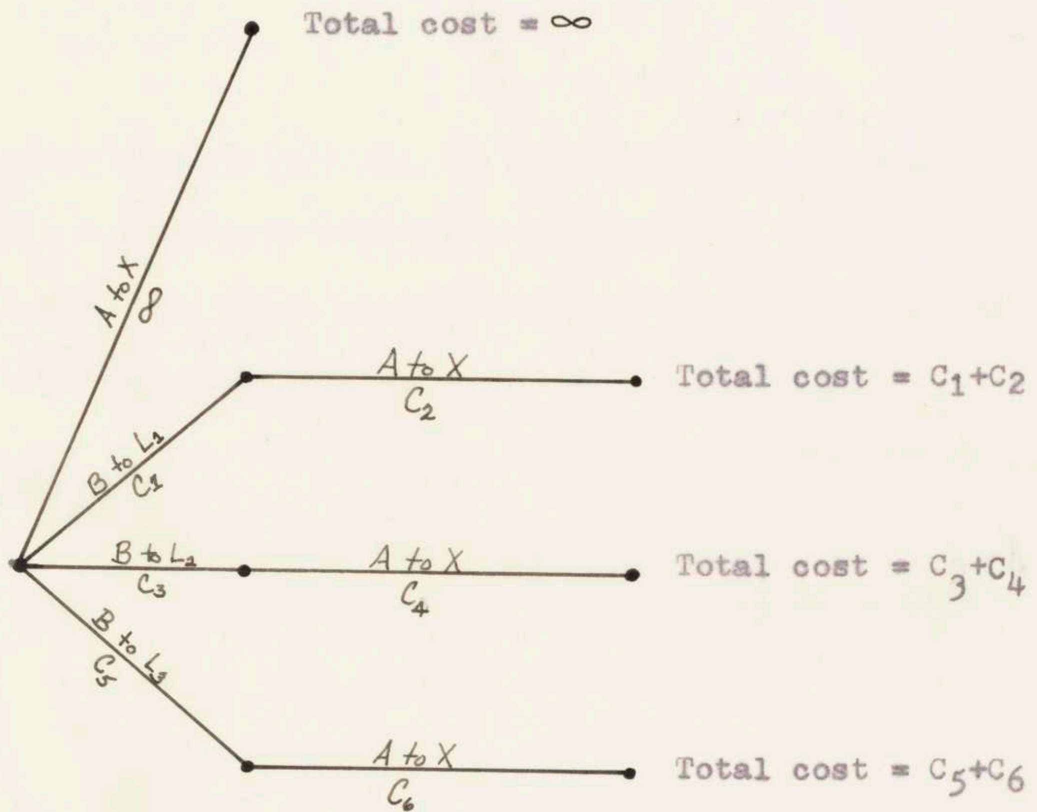
Task: move A to X

Operator must tell the system to move B, and give the system an area where it can be moved.

Hatched objects are immovable.

Figure 3

OR TREE to solve task shown in figure 3.



C's are costs

Figure 4

system to find the solution to the requested task, and

2) the system must compute a large number of paths.

This proposed OR TREE system of Whitney's will be discussed further in Chapter VII.

The Requirements of a Supervisory Controlled Manipulator

For a supervisory controlled manipulator to solve manipulation tasks, there is a minimal amount of information it must be given about the task, and there are certain abilities the system must possess. These are summarized below.

Input:

1) It must be given a description of the present task site so that it can make an internal model.

2) It must be given a description of the state of the task site as the operator would like it to be.

System:

1) It must have the means to change the state of the task site.

2) It must know *how* it can change the state of the task site.

3) It must have a method of planning changes which result in the solution of the requested task.

The goal of researchers in this area is to improve the system's abilities so that more complex tasks can be solved while the input is kept at the minimum. One measure of a supervisory controlled manipulator system, then, is the complexity of the task it can solve when given the minimal input described above.

Similar Work

Several other researchers have investigated problems of building supervisory controlled manipulators. Two notable examples are mentioned below.

The SRI Robot project[21] has, as one component, routines which plan and direct the robot to move objects around a room. The operator communicates with a question-answering system, QA3.[10] He phrases his requests in terms of conjectures which QA3 proves logically, using a predicate calculus technique called "resolution." For example, if the operator wants to move object X to position P, he conjectures (in logical terms) that a situation exists in which X is at P. QA3

then proves that this situation can exist, if the robot pushes X to P. The system traces through the proof to find which objects are to be moved (X in this case) and where they are to be moved to (P). The system then calls the path planning routine, plans the path, and executes it. The path planning routine utilizes the shortest path algorithm by Hart et al.[11]

Hewitt's PLANNER[12] is a programming language which is oriented toward accomplishment of tasks or goals which may be broken down into sub-tasks or sub-goals. The data, or theorems, needed to accomplish a desired result need not be referenced explicitly but rather by requesting, in essence, "the datum or procedure which accomplishes the desired result." This is like having the ability to say "Call a subroutine which will achieve the desired result." For example, if a theorem, T, is to be proved, PLANNER is asked to evaluate "GOAL T." PLANNER also has a back-up feature which allows it, in the case of failure, to return to the last place where a decision was made, make another decision at that point and continue searching. This feature allows PLANNER to explore a sub-goal tree; other recursive evaluators, like LISP, have no convenient way to do this.

For PLANNER to plan paths for moving objects, the operator asks PLANNER to prove that an object, X, could

be in a position, P. If its data base includes the appropriate axioms, PLANNER, after exploring the necessary subgoal trees, would find that object X could be in position P if it were moved there. Note that although PLANNER seems to work like QA3, the methods used by the two systems are very different.

As manipulation tasks can be posed in logical terms, any logical problem-solving machine can be used to find solutions. Notable examples in addition to the two previously mentioned include the General Problem Solver[19] and the Logic Theory Machine.[18]

Discussion of the Problem

Manipulation tasks generally are not difficult for people to solve. But there are instances where people cannot, or do not desire to, solve them. In these cases, people rely on mechanical servants. A mechanical servant that (potentially) can perform general manipulation tasks is commonly called a robot; in this work it is called a supervisory controlled manipulator.

The design of a supervisory controlled manipulator poses many difficult problems. Three general problem areas that seem to be most important are:

1) Design of the hardware (includes computers, vehicles, hands, and arms).
2) Design of a system (computer program) which can understand a task request made in the operator's natural language.
3) Design of a system (computer program) which can figure out how to accomplish the task request.

This thesis concentrates on the third aspect of the problem. Our goal is to design a planning procedure, which, when provided only with a description of the task site and a task request that is strictly limited to <u>what</u> the operator wants done, will have the ability to find good solutions to complex manipulation tasks. (Complex manipulation tasks are defined as manipulation tasks in which two or more objects are to be moved.)

At this point some general comments are in order. First, for the present time we will restrict our attention to manipulation tasks in which we are concerned only with the static arrangement of objects. An object being moved by the jaws will have a velocity, but we are interested only in the fact that the object and the jaws occupy a succession of specific states as they are moving.

Second, we are interested in finding good, not necessarily optimal, solutions. Optimal techniques are used, and path segments will be optimal in terms of an a priori cost function, although the overall solution will not be. The objective is to find a string of manipulator primitives which, when executed in order, give a solution to the requested task without much wasted motion.

Third, the reader should realize that there will be limits on the capability of the system proposed in this thesis. The goal is to increase the degree of complexity of the manipulation tasks that can be performed when given the minimal input. There will be a discussion of the limits of this system in Chapter V.

Preview of the System Proposed in This Thesis

The discussions and descriptions in this thesis are to be at three levels:

1) general task or activity planning,

2) manipulation task planning, and

3) examples used for illustrations.

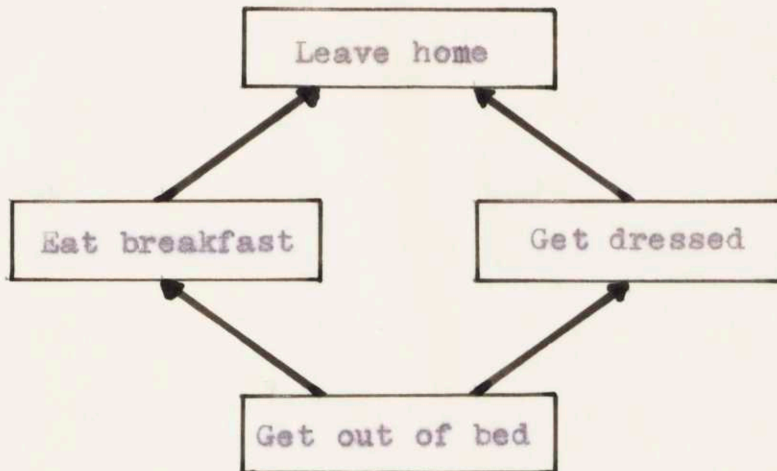
A characteristic or ability described at one level can be assumed at any lower, more job specific level. But any

restrictions on low level abilities must not be assumed to be valid at a higher level.

At the general level, there will be a discussion of the characteristics and uses of AND TREES. AND TREES serve the same function as PERT charts; and they can be used interchangeably, but they are not the same. A PERT chart orders the sub-activities of an activity so that one can accomplish it by executing all the sub-activities on the PERT chart in the indicated order. A plan made in PERT chart form can be put on an AND TREE by replacing the duplicate edges of the PERT chart with duplicate nodes on the AND TREE. Like a PERT chart, all tasks on an AND TREE must be executed. The order of execution is from bottom to top. The following example will illustrate the above ideas.

Suppose we plan getting ready to go to work. The first step is to get out of bed, then dress, eat breakfast, and finally leave. Getting out of bed must precede the other events, and leaving home must follow all other events. Getting dressed and eating breakfast can be done in any order. A PERT chart diagramming these activities is shown in figure 5. The same plan on an AND TREE is shown in figure 6. The duplicate edges of the PERT chart are replaced by duplicate nodes on the AND TREE.

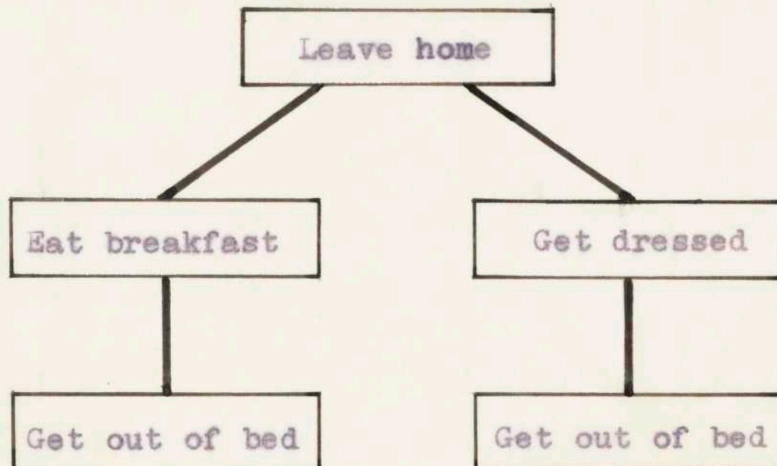
PERT chart plan for getting ready to go to work.



All tasks must be executed in the order indicated by the arrows.

Figure 5

AND TREE plan for getting ready to go to work.



All tasks must be executed from bottom to top.

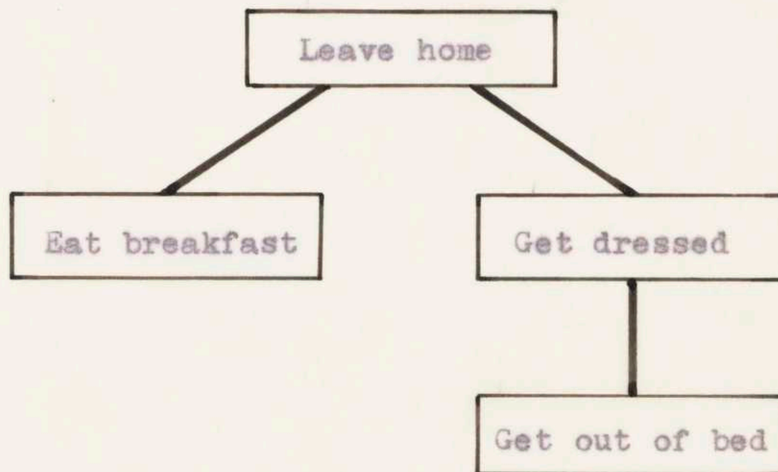
Figure 6

If we go one step further and adopt a direction of execution convention (the direction is arbitrary, but once it is chosen, it must be adhered to), we can reduce the nodes on the AND TREE. Figure 7 shows the AND TREE of figure 6, after adopting a right to left execution convention, and deleting duplicate tasks.

AND TREES are used because they are easier to implement on a computer, as each node has one, and only one, predecessor node on the tree. As they are equivalent data structures, the advantages are available without penalties.

To find the solutions to complex manipulation tasks, Whitney's state space technique will be used to solve the simple manipulation tasks (move one object), and the AND TREE will be used as a data structure to order the simple tasks. The result of the calculations is a string of manipulation primitives which, when executed in order, will move objects to the requested final positions.

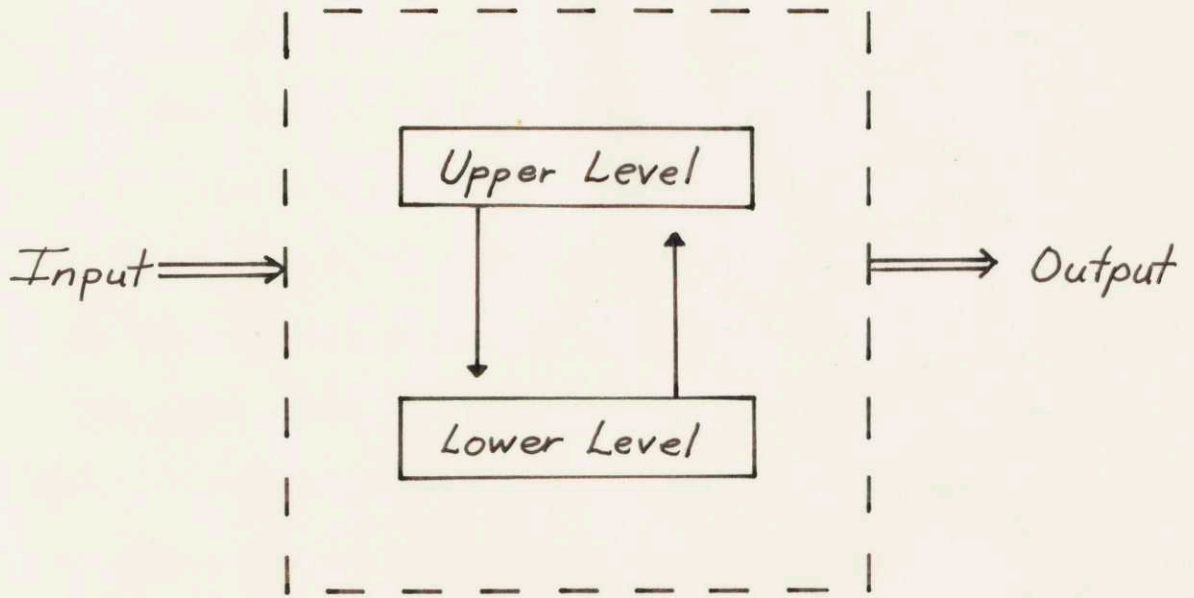
A simplified block diagram of the system is shown in figure 8. The input consists of the objects' shapes, initial positions, and specified final positions. In the system implementation, these inputs are given by teletype, but are essentially descriptions of snap shots of the desired configuration. (See the pictures in Chapter VIII.)



AND TREE plan for getting ready to go to work after adopting a left to right execution convention and deleting the duplicate tasks.

Figure 7

Simplified Block Diagram of the System



Input: objects' shapes, initial positions, and specified final positions

Output: path for jaws and objects

Upper level of system: AND TREE to order sub-tasks

Lower level of system: a state space method to find solutions to sub-tasks

Figure 8

Note that only one object must have a specified final position for a task to be defined; that is, not all objects on the initial position list must be on the final position list. For these other objects the system assumes the operator does not care where they are at the end of the task. It tends to leave these objects scattered around the space. Note the position of objects B, F, and D in the last picture of task 1 in the examples section, frame 211.

The lower level of the system is an implementation of a shortest path algorithm that is designed to plan a path for moving <u>one</u> object. It is, in essence, Whitney's state space method, but the algorithms are implemented differently to handle random shaped objects. Chapter IV discusses the implementation of the shortest path algorithm.

The main part of the upper level of the system is an AND TREE which determines the order for moving objects to their final positions. The upper level also generates and orders the sub-tasks which specify moving objects out of the way of the proposed path of others.

The output of the system is a path for the jaws and objects which shows the step by step changes in positions. After the last step of the path has been completed, the task as defined by the operator is completed.

Briefly, the system works as follows. The system sets up a model of the physical space where the task is to be executed, and the operator gives it a description of the task space as he would like it when the task is completed. The system uses this information to generate an order to move objects to their specified Final Positions. This order is represented by a stack of sub-tasks (each of which requests moving one object) on the AND TREE. The upper level system gives the lower level system the name of an object to be moved, and a position to which it is to be moved. The lower level system attempts to compute a path for the object and returns to the upper level system a value corresponding to:

1) A path is found and the object can be moved.
2) A path is found but other objects are in the path and must be moved out of the way before this object can be moved.
3) No path is found to move this object.

If the value corresponding to 1 is returned, the object in the system's internal task model is moved. If the value corresponding to 2 is returned, sub-tasks are generated to move the objects in the planned path out of the way. These sub-tasks are put on the AND TREE to be executed prior to the sub-task whose execution was just attempted. If the value

corresponding to 3 is returned, the shortest path algorithm failed to find a path (for example, the task is impossible because of an immovable wall), and the system terminates execution of this entire task.

When values corresponding to either 1 or 2 are returned, the upper level system gives the lower level system another sub-task to execute. This sequence continues until the system finds a solution to the complete task as defined by the operator, or until it discovers, using its internal model of the task, that task is impossible.

As an example, consider the following task. Figure 9-a shows the objects' Initial Positions and figure 9-b shows their Final Positions. The small square with the crooked line ("W") through it is the manipulator jaws. They are the prime movers; they move the objects in the space by grasping or pushing them.

The system decides that the order in which the objects are to be moved to their final positions is A, then B, and then C. (There are only two other orders that are possible: B, A, and C; or B, C, and A.) The system's solution is as follows: the upper level system asks the lower level system to find a path to move A to its final position. The system then discovers that B is in the way. (A is not moved.) The

Sample Task Specification

Initial Positions

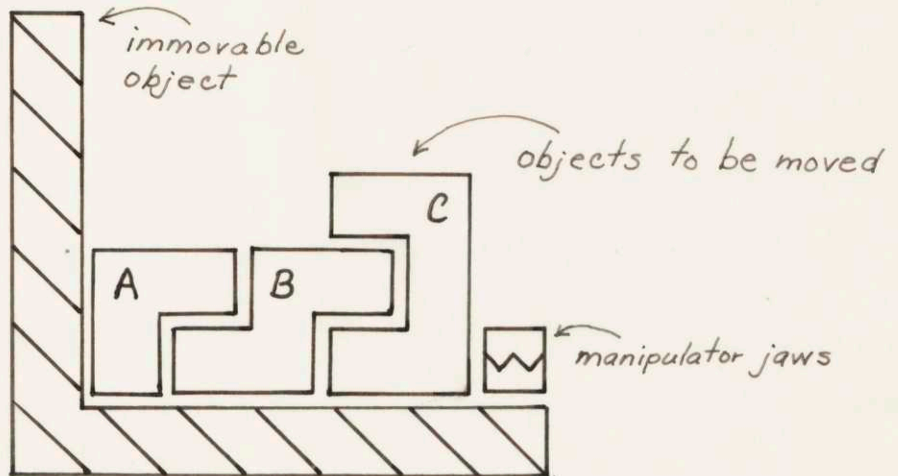


Figure 9-a

Final Positions

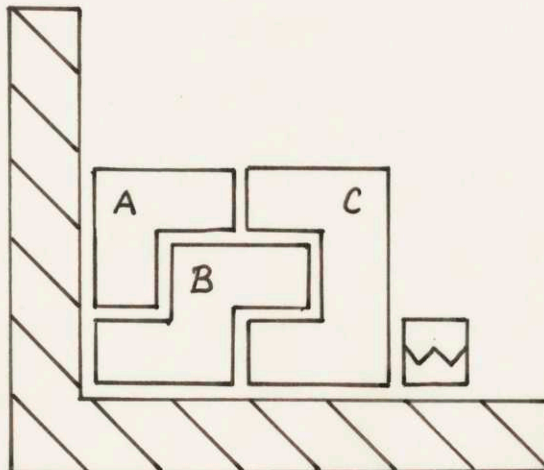


Figure 9-b

system tries to move B out of the way and discovers C is in the way. The system then finds a path in which the jaws push C out of B's and A's way, then finds the next segment of the path which moves B out of A's way. At this point B and C are out of the way, and the objects may be moved to their final positions. The system continues and finds the path for the jaws to move A to its final position, then B to its final position, and finally C to its final position. The last segment of the path directs the jaws to move to their final position. (For further examples, the reader is referred to Chapter VIII.)

Preview of the Remainder of This Thesis

The next chapter contains an explanation of an AND TREE, what some of its properties are, and how it can be used to order the sub-activities of a general activity. Chapters III and IV contain comments about the lower level system, the shortest path algorithm, and the particular implementations used for this work. Also included in Chapter IV are descriptions of the restrictions on the demonstration system and outlines of some basic manipulation situations. Chapter V describes some of the situations in which the system will fail to find solutions, explains why it fails,

and suggests remedies to overcome the failures. It also includes explanations of how the system finds solutions (if solutions exist!) of explicitly defined task types. Chapter VI describes the ability of the presently implemented system to diagnose failures. It also gives a brief outline of those diagnostic aids which could be added.

Chapter VII contains a comparison of the advantages of using three proposed systems:

1) A full optimization solution of the entire problem.

2) The OR TREE method proposed by Whitney.

3) The system described in this thesis.

The final chapters contain examples of tasks this system has solved, followed by suggestions of specific problems that need future work and the conclusions. Appendix C describes the input/output formats required by the program, and Appendix D contains a high level flow chart of the demonstration system.

Chapter II Basis for This System

The basis of this thesis is to use an AND TREE on which are arranged several simple manipulation tasks to represent a complex manipulation task. The solutions to the simple tasks can be found using a state space system like Whitney's,[31] and the characteristics of the AND TREE will assure that the order in which the simple tasks are executed will be an order which will give a solution to the complex task.

In this chapter the characteristics of an AND TREE, including the characteristics of the tasks that can be analyzed by an AND TREE system, will be investigated and an outline of how the AND TREE can solve complex manipulation tasks will be given.

Characteristics of an AND TREE

An AND TREE is composed of nodes and edges. For our purposes, the nodes represent sub-tasks, and the edges connect the sub-tasks. Figure 11 is a drawing of an AND TREE. The AND TREE looks like the familiar OR TREE, or decision tree, but it differs in that

1) all branches must be executed, and

2) execution begins at the bottom of the branches and proceeds toward the top of the tree.

Because there is a direction of execution of the nodes on the tree, the AND TREE can order the execution of a set of sub-tasks.

To aid in the following discussion, the terms predecessor node, successor node, and brother node will be defined. A predecessor node is closer to the top of the tree and on the same branch as a given node. In figure 11, nodes 8 and 4 (the upper node 4) are both predecessors of node 10. A successor node is closer to the bottom of the tree and on the same branch as a given node. In figure 11, node 11 is the successor of nodes 8 and 4 (the upper node 4). Brother nodes have the same immediate predecessor. In figure 11, nodes 7, 8, and 9 are brothers, but nodes 6 and 7 are not brothers.

The AND TREE is a useful way of specifying in varying amounts of detail, any activity which can be divided into sub-activities. The TREE provides a structure which can specify hierarchical relationships. It also can keep lists of sub-activities that have no hierarchical relationship. And it can be used in situations where the hierarchical relationships between some sub-activities are important and other

sub-activities have no important hierarchical relationship to one another. To illustrate, consider the AND TREE shown in figure 11 (disregard the lower sub-tasks 2 and 4). Sub-tasks 10, 11, 12, and 13 can be executed in any order, but all of them must be executed prior to sub-task 8.

The familiar data structure that does the same job as the AND TREE is the PERT chart. The PERT chart is a graph, but not a tree as it generally contains too many edges. A graph which contains n nodes is defined to be a tree if the graph is connected and if there are $n-1$ edges. A tree must contain at least two nodes. There has been much written about PERT charts, both theory and use. See, for example, Archibald[1] and Shaffer et al.[26]

The AND TREE is a special case of a graph. There are several books on graphs which have some sections devoted to the characteristics of trees. The interested reader is referred to Berge,[3] Ore,[22] or the NASA Technical Report 32-1413[4] which is a detailed review of the literature available on graphs.

The OR TREE can also specify the order in which sub-tasks are to be executed. But, because only one branch of an OR TREE is executed, all sub-tasks must be on all branches. The OR TREE with all possible permutations of n sub-tasks

of a complex task has n! branches and $\frac{n \cdot (n!)}{\prod_{i=1}^{n} [(i-1)!]}$ nodes. An AND TREE, on the other hand, need have only n nodes and $\frac{n+2}{2}$ or fewer branches.

To be analyzed by the system, a complex task must be made up of specific sub-tasks; that is, the sub-tasks must request an action be performed on or with a specific object, or at a specific location. An example of a specific task is "Move object A to position X." An example of a non-specific task is "Move an object." If this requirement is not met, the system cannot detect loops in the task structure (loop detection will be discussed later in this chapter).

A second requirement a complex task must meet is that the system must be able to figure out how to perform the sub-tasks of the complex task, or that the system be pre-programmed to perform the sub-tasks. Also, the system could break the sub-tasks up into sub-sub-tasks, etc. But at some point the system must know, or be able to figure out, how to perform the sub-tasks.

The ability to analyze sub-tasks must include the ability to determine if another sub-task should be performed prior to the sub-task whose execution is being contemplated. The system must have this ability if it is to

figure out that the order implied by the direction of execution of brother tasks on the AND TREE is not the one that will achieve the requested complex task. To use this information the system must be able to add sub-tasks to the AND TREE.

A third requirement of complex tasks is that the sub-tasks, when executed <u>one at a time</u> in some order, will give a solution to the complex task. This requirement eliminates those complex tasks whose solution requires that two or more sub-tasks be executed at the same time. An example of such a task is one that requires assembling a spring loaded mechanism. One typically has to compress the spring, put a plate in place, and insert and start two or three bolts. The spring must be held in place while the last two sub-tasks are executed; two sub-tasks must be executed simultaneously.

A general AND TREE system would work in the following way. All sub-tasks that are explicitly requested are put on an AND TREE. Then the system either breaks the tasks into sub-tasks or figures out how to perform them. Generally, the system will discover that one or more other sub-tasks must be performed before the requested task can be performed.

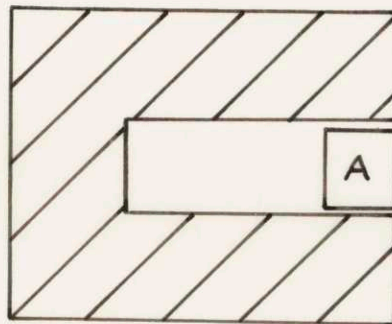
The system then puts these other sub-tasks on the AND TREE, prior to the task whose execution was planned.

The system continues, trying to execute the sub-tasks at the bottom of the AND TREE. When a sub-task can be executed (without requiring the execution of any other sub-task) that branch of the tree is terminated. The above sequence is continued until all branches are terminated. This will be demonstrated in an example to follow.

Besides the branch termination where the task can be performed, there are two other possibilities for termination. The first is a case in which a task is impossible to perform. In the case in which the impossible task is explicitly requested by the operator, the system will inform the operator that the task is impossible. Figure 10 is an example of an impossible two dimensional task. In the case where the impossible task is generated by the system, the system will try to find other tasks that can be performed which, when executed, allow the task requested by the operator to be performed.

The other possibility for termination of an AND TREE branch occurs when the system detects a task loop. The AND TREE gives one the opportunity to discover, quite easily,

Task: move A to X



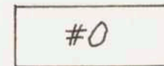
An impossible two dimensional task

Figure 10

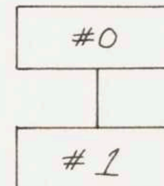
if an activity is its own predecessor. In the case where an activity is its own predecessor, either immediately or several predecessor nodes up the tree, there is said to be a <u>loop</u> in the activity structure (that is, on the TREE). Figure 11 shows an AND TREE with two loops. The ability to detect loops easily is valuable as the number of sub-activities in a complex activity may be very large.

The concept of how a task loop is generated and detected is illustrated in the following example.

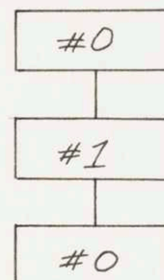
The system is requested to perform task #0 (it does not matter what task #0 is). Task #0 is put on the first level of the AND TREE.



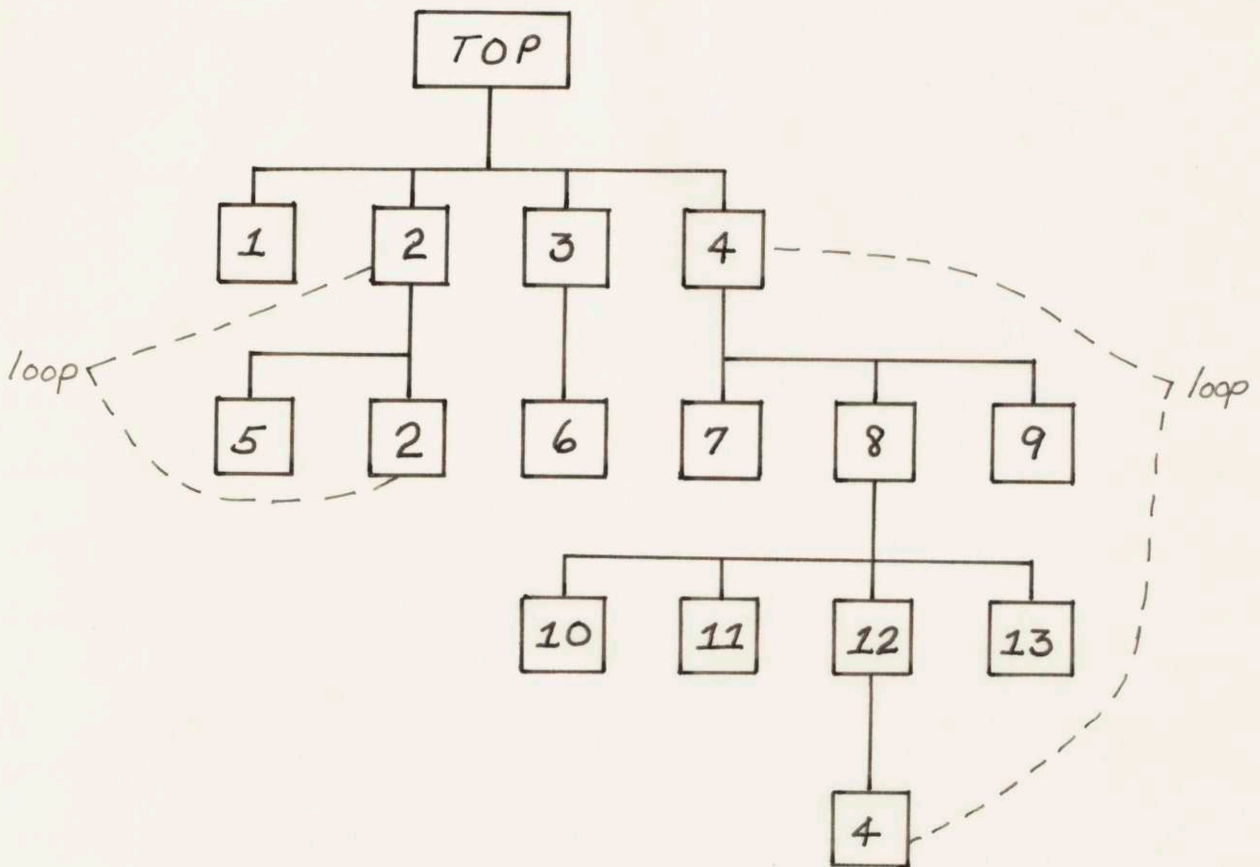
In planning to execute this task, the system finds that the best thing to do is to first perform task #1. Task #1 is then put on the AND TREE prior to task #0.



The system now plans to perform task #1. It finds that the cheapest way to execute this task is to first perform task #0. The system then puts task #0 on the AND TREE prior to task #1. There now has developed a <u>task loop</u>, as a task is its own predecessor.



An AND TREE



All branches must be executed

Loops are shown involving tasks 2 and 4

Figure 11

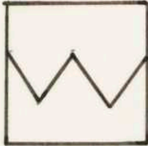
To stop the system from forming the same task loop again, a special note is added to the task (on the TASK TREE) that comes just before the second instance of the task which constituted the loop (the note is added to task #1 in this case). This note says that when the environment is in the present configuration (which must be remembered), it costs an infinite amount to perform task #0 when planning to perform task #1. If there is no alternative to a loop, the system will respond that the task requested is impossible (costs an infinite amount to perform).

As an example of how the system detects loops when performing an actual task, consider the task depicted in figure 12. As shown, the requested task is impossible. The system decides this in the following manner.

First the system plans to move A to X. It finds that first, B must be moved. (How the system finds this will be discussed in Chapter IV.) So it plans to move B to an Out of the Way Place. In making this plan, the system finds that A must first be moved. The system now discovers that this AND TREE has a loop by finding that the task just added is its own predecessor. (Move A - Move B - Move A). The system then makes a note that when the environment is in its present configuration A should be considered a fixed

Impossible two dimensional task demonstrating a task loop

Task: move A to X



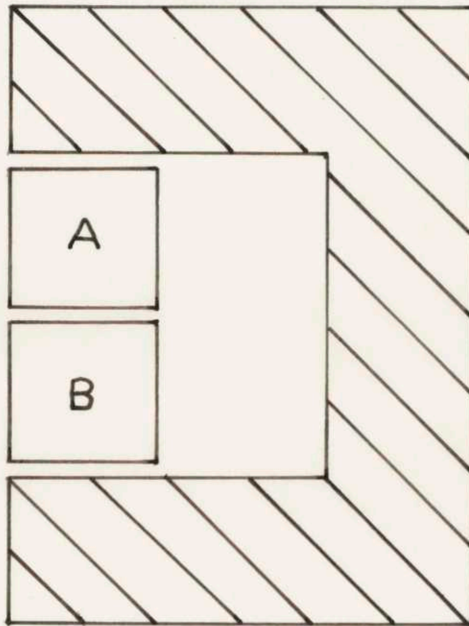


Figure 12

object when planning to move B. When the system plans to move B out of the way the next time, it finds that the sub-task is impossible, and hence, that moving A is impossible.

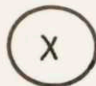
The TASK TREE

For the purposes of this thesis, the AND TREE will be referred to as the TASK TREE. An example of a TASK TREE is shown in figure 14. To show its usefulness, let us consider the following example. Suppose we have the situation shown in figure 13. The lettered objects are movable blocks. The task assigned is to move block A to position X. This is a two dimensional problem, and objects can be moved only one at a time.

The solution is to move objects B and C out of the way, and then move A to X. The system discovers that B and C must be moved out of the way, generates the sub-tasks to request this, and puts the sub-tasks on the TASK TREE prior to the sub-task to move A to X. The complete TASK TREE for the task request is shown in figure 14.

If there were other objects encumbering the motion of B and C, then requests to move them to Out of the Way Places would be shown on the TASK TREE prior to the requests to move B and C (just as the requests to move B and C are

Task: move A to X



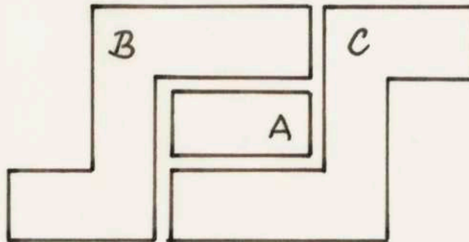
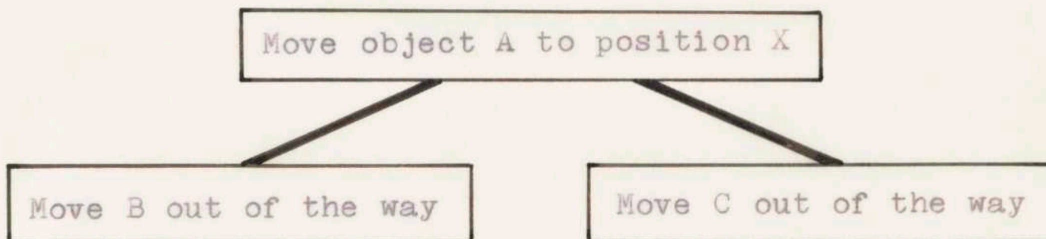


Figure 13

TASK TREE for task shown in figure 13



The system generates the sub-tasks to move B and C

Figure 14

TASK TREE and Interpretive List are the same as TASK TREE in figure 14

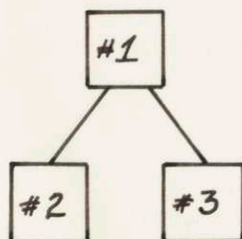


Figure 15-a

<u>Interpretive List</u>

Task #1	move A to X
Task #2	move B out of the way
Task #3	move C out of the way

Figure 15-b

prior to the request to move A).

For ease of operation in the proposed system, the tasks are given number codes--task #1, task #2, etc. These numbers are put on the TASK TREE, and an interpretive list is made telling which task corresponds to which number. This list can be referenced from either a task number or a task name. The TASK TREE shown in figure 15-a with the interpretive list of figure 15-b is the equivalent of the TASK TREE of figure 14.

The part of the TASK TREE system that detects failures can be fairly simple. Manipulation tasks (in which we are interested) are requested by specifying a goal state which differs from a given initial state. To determine if it has failed to solve the requested manipulation task, the TASK TREE system only has to compare the final state of its internal model to the requested goal state.

A TASK TREE is a workable way of making plans for complex tasks that consist of many sub-tasks. It can store (in an orderly manner that allows computer processing) the important aspect of the task that we need for making plans for performing large, complex tasks--which tasks must be performed before other tasks. A TASK TREE can maintain cost

information about all tasks so one can know not only the order to perform tasks, but also how much they will cost.

The TASK TREE system must be able to request solutions to simple tasks from a lower level system it supervises, by giving the lower level system the information it needs to find the solutions to the simple tasks. This higher level system does not know anything about the nature of the tasks it is processing; it is concerned only with abstractions. It only knows about the TREE, the names, as it were, for the simple tasks on the TREE, and whether the lower level system finds solutions for the simple tasks.

The TASK TREE structure allows a computer to determine if there are any logical task loops in the tree. This ability to detect loops gives the user confidence that the system will not perform a large number of tasks before an operator discovers it is in a loop.

Also, the TASK TREE structure allows for dynamic growing and shrinking. It is not limited to any particular number or type of tasks. "Given a larger computer... ."

Chapter III <u>Shortest Path Algorithms</u>

This chapter is an explanation of the shortest path algorithms used in the demonstration system. The reader who is not interested in the detailed workings of these algorithms may go to the next chapter with no loss of continuity.

Before beginning the explanations of the algorithms, some terms which will aid in the discussions need to be defined. The algorithms will usually begin from one point, the <u>start</u>. In a state space, the <u>starting point</u> is the state (point) that represents the current configuration of the physical space. Likewise, the <u>finish</u> is the point to which the algorithms will find a path. In a state space, the <u>finishing point</u> is the goal state (point) that represents the desired configuration of the physical space.

To <u>expand</u> a point in the state space is to investigate the total cost to get from the start to each of this point's neighbors with the path including this point as the immediately preceding point. That is, the last two sections of the flow chart, figure 16, are the "expansion of a point." (Note that it is not possible to get from each point in the state space to all other points in one step. In a two dimensional space with a rectangular grid of points in one step

it would be possible to get from one point only to its four nearest neighbors (no diagonal transition is allowed). This implementation of the algorithms considers only the possibility of transitions from a point to its nearest neighbors, however, the descriptions of the algorithms' properties will be general so that the reader can apply these algorithms to cases in which transitions are possible to other than the nearest neighbors.) The *front* is the set of points to which the algorithm has found at least one path, but has not yet decided that it has found the cheapest path. That is, the points on the front are neighbors of points that have been expanded, but they (the points on the front) have not yet been "*closed*." To *close* a point is to set a flag that indicates that the cheapest path to this point has been found, and there is no use trying to find others. The *cost of a point* is defined as the cost to get from the start to the point. Similarly, the *cost of the front* is the cost to get from the start to the lowest cost point on the front.

Flooding Algorithm

The basic flow chart for the flooding algorithm is shown in figure 16. The flow chart is complete, except for the various termination conditions we will want to utilize.

FLOODING ALGORITHM:
Basic Flow Chart

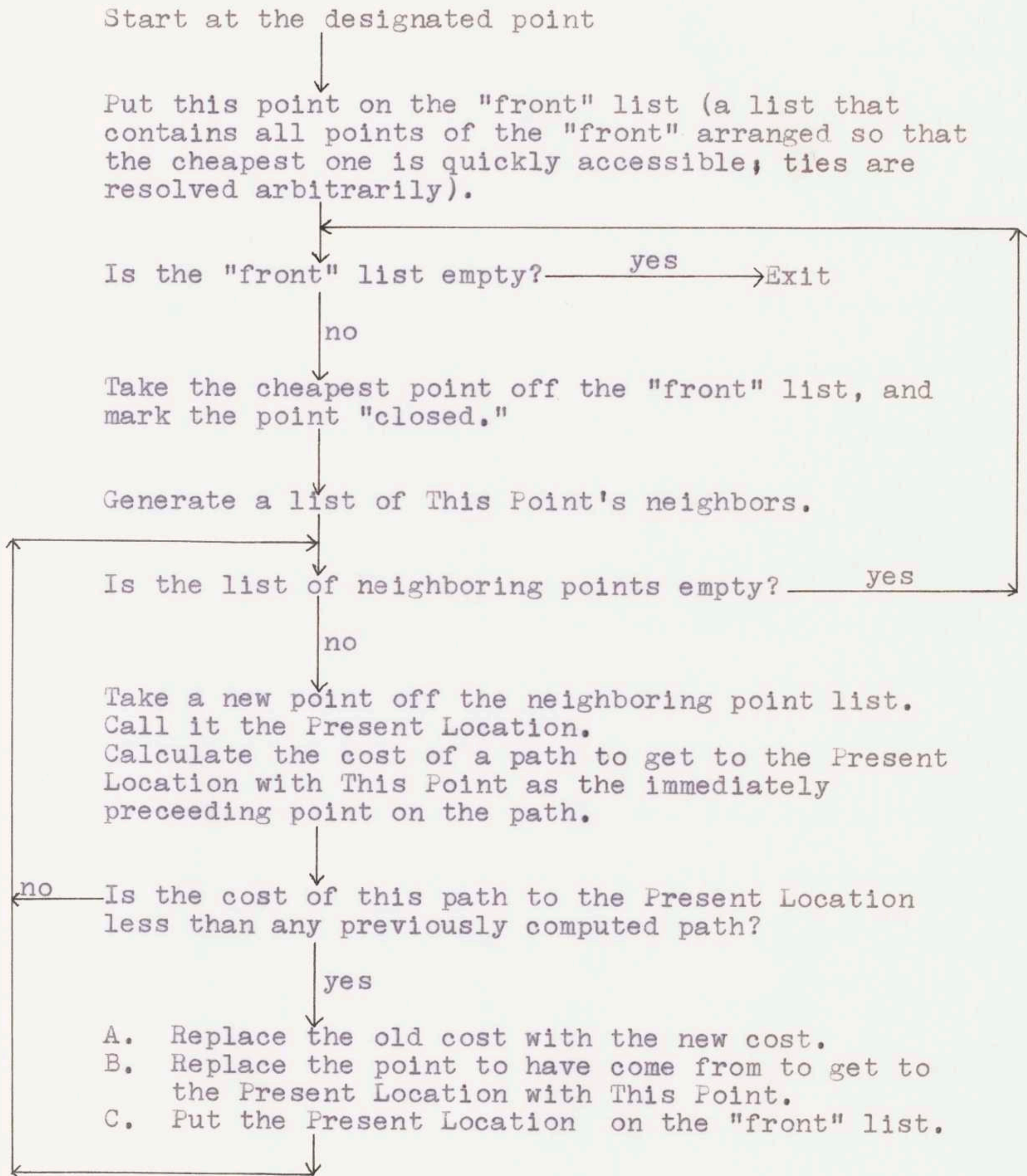


Figure 16

These termination conditions will be added as they are discussed.

The point we choose to expand each time is the lowest cost point on the front. Doing this guarantees us the following relation: the cost to get to the lowest cost point on the front is greater than or equal to the cost to get to any point inside the front (inside and outside the front as defined below). This relation holds because we previously expanded and closed the points inside the front, hence the cost to get to them is less than or equal to (in the case of ties) the cost to get to the lowest cost point on the front. There is also another relation we can use to our benefit: the cost to get to any point outside the front, when found, will be greater than the cost to get to the lowest cost point on the front. This relation holds because the minimum cost transition from one point to another is greater than zero, and the path from the start to any point outside the front must include, when found, at least one member of the present front. These two relations can be summarized as: The cost to get to any point outside the front, when found, will be greater than the cost to get to the lowest cost point on the front, which is greater than or equal to the cost to

get to any point inside the front. The following example will clarify the concepts of "inside" and "outside" the front.

The initial situation is as shown in figure 17, except that no points are shown other than the starting point and its four nearest neighbors. The other points in the space lie in the same rectangular grid pattern as the five points shown, and extend in all directions.

We will call the center point of figure 17 the starting point and set its cost to zero. This point is now put on the front list. This is the only point on the front. All other points in the space are outside the front. Initially, the cost to get to these points is infinity, or some number large enough so that all necessary relations will hold. (A good number is 10 times a maximum dimension of the space times the maximum transition cost in the space. For computer implementations, the largest possible positive number works very well.) At this stage, there are no points inside the front.

The cost to make a transition from one point to another must be greater than zero. For convenience, let all transition costs be integers, with the minimum cost transition being one (1).

Initial Stage
Finding a path through a space using the flooding algorithm

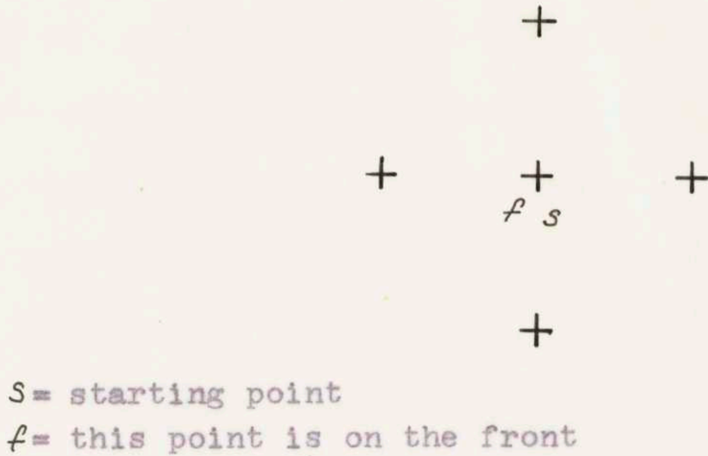


Figure 17

Second Stage
Finding a path through a space using the flooding algorithm

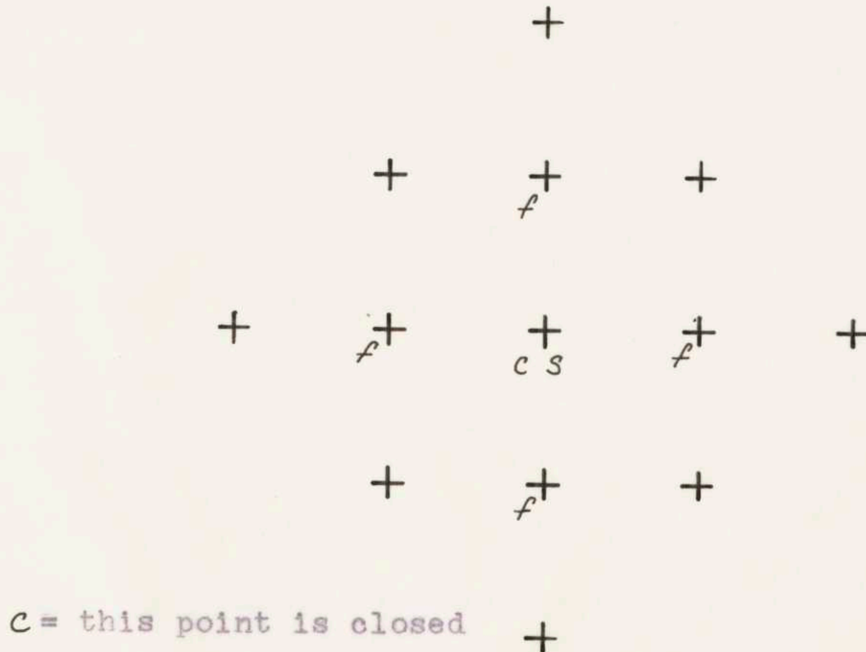


Figure 18

We now pick the lowest cost point off the front list, mark it closed (this is the starting point; there can be no cheaper way to get here), and expand it. This point's neighbors are now on the front list (they were put there by the algorithm after it found the paths and costs). The situation is now as shown in figure 18.

Let the algorithm continue. We pick the lowest cost point off the front (resolve ties arbitrarily). Then we mark this point as closed; its cost cannot decrease. Let us see why this is so.

The points in the space can be classified as belonging to one of three sets: points inside the front, points on the front, and points outside the front. Let us consider the possibilities that a lower cost path to this point could be found. First, consider all the points inside the front. In this case there is only one point; and the path includes it as one of the two points on the path. (The path consists of this point and the starting point.) Hence, there is no need to reconsider this alternative. Next, consider the points on the front. In picking the lowest cost point on the front (remember the possibility of ties), we guarantee that the cost of all other points on the front is greater than or

equal to the cost to get to this point. Hence, any path to this point which includes any other point on the front cannot possibly be cheaper than the path that the algorithm has presently settled on (all costs are greater than 0).

Finally, there are the points outside the front. We have not yet found a path to them from the start, and therefore the cost to get there is presently infinity. However, when we do find a path to each of these points, the cost of this path must be greater than or equal to the cost to get to the lowest cost point on the front as the path must include at least one point that is now on the front. Hence, any path from the start that includes one of these points must be more expensive than the path to the lowest cost point on the front.

To resume the discussion, we just picked the lowest cost point off the front and marked it closed. We then expand the front to include the points we can get to from this point where this path is cheaper than any previously computed. The situation is now as shown in figure 19.

Now let us examine the situation in figure 19. There are two closed points. We have shown that we have found the <u>cheapest</u> way to get to them from the start. There are a number

Third Stage
Finding a path through a space using the flooding algorithm

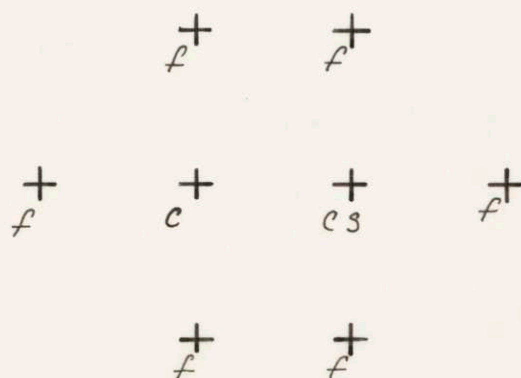


Figure 19

of points on the front whose costs are greater than or equal to the cost to get to any point inside the front. There are also the points outside the front, (again not shown) and we know the cost to get to any of these points is presently infinity.

Now, we will pick the lowest cost point on the front, and mark it closed; the cost to get from the start to this point cannot decrease. Again, we will pause to see why this is so.

There are again, the three sets of points: points inside the front, points on the front, and points outside the front. There are two points inside the front. Each of them has previously been on the front, and we have checked the cost to get from each of their neighbors, through them, back to the start, and retained the lowest cost computed for each instance. Hence, we have already considered all the possibilities of lowest cost paths that are possible from points inside the front. And we have retained the lowest cost of these. There is no reason to reconsider these possibilities. If we do, the lowest cost path we could possible find would be this path, or possibly another path of equal cost. This argument is general. It applies when there are any number of points (more than zero) inside the front.

The arguments previously given as to why we cannot find a lower cost path from this point to the start through any of the points on or outside the front continue to hold. (Note that the path to the lowest cost point on the front can include <u>only</u> points inside the front.)

Again, to resume the explanation. We just picked the lowest cost point off the front and marked it closed. We then expand the point and increase the front list by the additional points we can get to from this point in the cases where the path found is cheaper than any previously computed. Now we can continue expanding the front, confident that it is finding the lowest cost paths until we reach a terminal point, which terminates the search.

Termination of the Search

The reason for using this search technique is its speed. It finds the shortest path to a point quickly if the cost of the path to the terminal point is less than the average of the costs of the paths to all the points in the space.

In the case of a single terminal point, the search terminates after the terminal point is closed. The sequence of events for the algorithm is: pick the lowest cost point

off the front list, mark it closed, and check to see if it is the termination point. If it is, the search is finished. If not, continue the normal way.

When there is a set of possible terminal points (we must find a path to <u>any one</u> of the set, the OR condition of multiple terminal points), the termination condition is the same, except that the process terminates as soon as any one of the points of the terminal set is closed. See figure 20.

How are we sure that this gives us the shortest path to <u>any</u> of the terminal set? The reason is basically the same as that used to justify closing the point. The cost to get to any open (not closed) points in the space will be greater than or equal to the cost to get to this point. And, as this is the first point of the terminal set to be closed, the cost to get to this point is less than or equal to the cost to get to any other point of the terminal set.

When there is a set of required terminal points (we must find a path to <u>all</u> of the points of the set, the AND condition of multiple terminal points), the algorithm terminates only after all points in the set have been closed. See figure 20.

FLOODING ALGORITHM:
Flow chart for termination with an OR set of terminal points

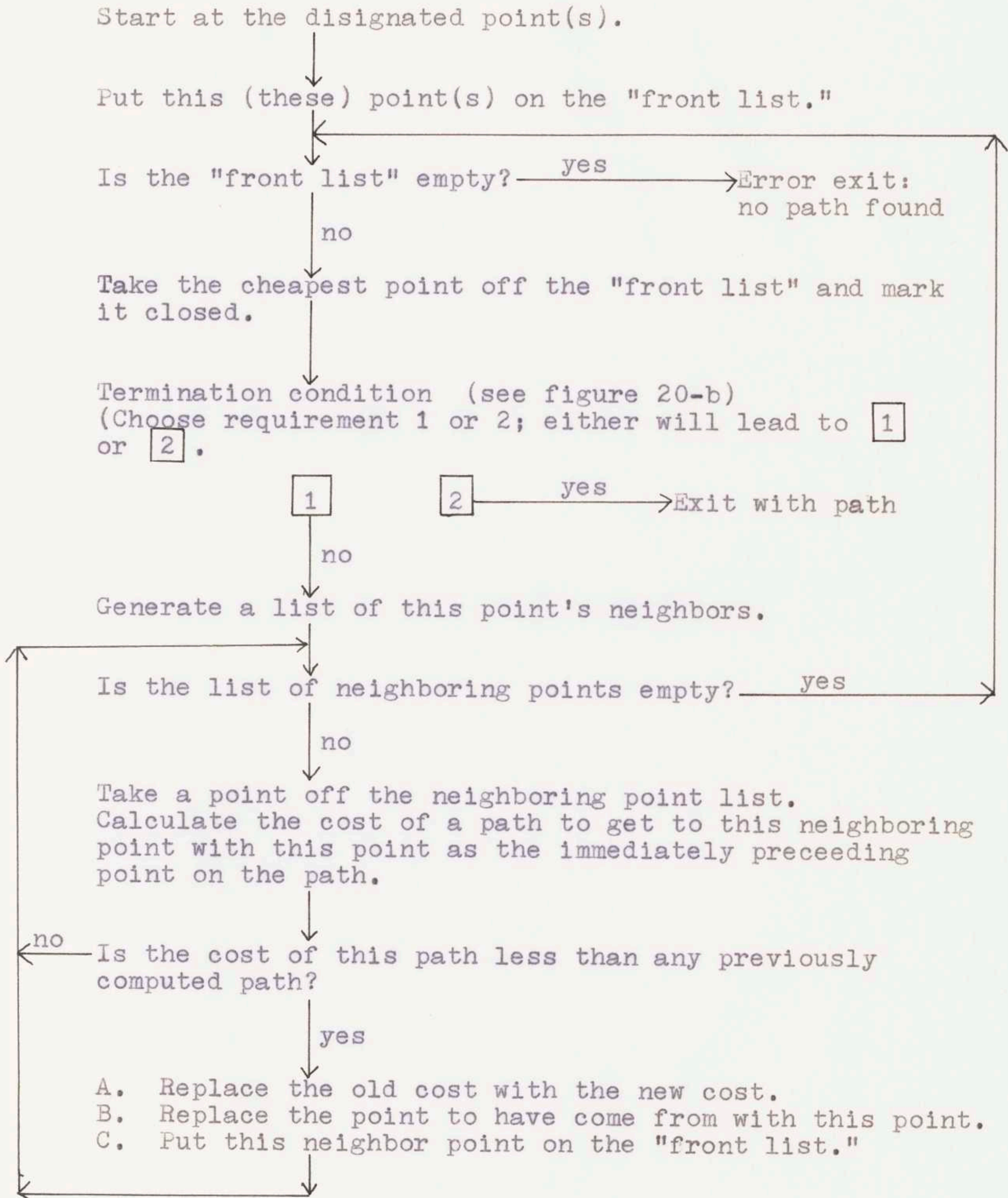
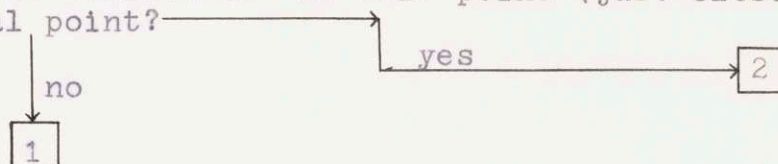


Figure 20-a

TERMINATION CONDITIONS:

1. Requirement: Must find a path to any one of several points.

   Termination Condition: Is this point (just closed) a terminal point?



2. Requirement: Must find a path to all of several points.

   Termination Condition: Is this point (just closed) a terminal point?

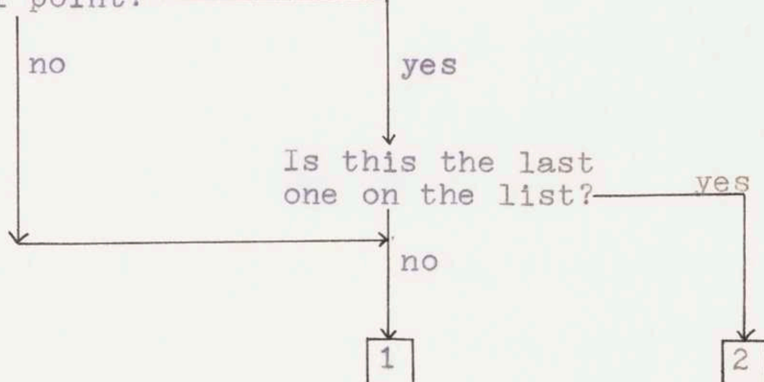


Figure 20-b

Starting From More Than One Point

The flooding algorithm will work as described previously when considering the possibility of starting from more than one point. The only change is that the algorithm should not set all starting points' cost equal to zero, but should allow the flexibility of different positive values for the starting costs.

An example will clarify. At the end of a working day you want to go home. However, you have a choice as to where you will be when you finish work, points A, B, or C. Due to circumstances other than closeness to home, you have preferences as to which of these you would rather be at when work ends. For instance, if at the end of work you are at B, you will be able to spend only 10 minutes there when you would like to spend 15. For this inconvenience, let us assign a cost of starting from B equal to 5 units. And say also that the cost of starting from A is 10 units, and from C is 2 units. Now, if the cost to get from A to home is 20 units, from B to home is 24 units, and from C to home is 29 units, a rational decision maker would pick B as the starting point. He would do this even if his route home takes him by A. (Note that given the costs, he should not go by C on the way home.)

This situation is the OR condition for multiple starting points. The implementation of this condition is quite easy. Calculate the cost to start at the various starting points, then put all the points on the front list and begin the calculation in the usual way. See figure 20.

If you look at the positions of the fronts after several iterations of the algorithm, the situation may be as shown in figure 21. Although there are actually several different lines that compose the front, they are all part of the same front. And, the relations we have previously proven still hold. If the algorithm is allowed to proceed, the "separate" fronts will join to form the more familiar pattern shown in figure 22.

Above we showed that the algorithm could start from multiple starting points (OR only) and finish with multiple finishing points (AND or OR). Now we want to introduce the possibility of starting the algorithm from two or more points. This is different from allowing the possibility of the path starting from two points. It is the AND condition for multiple starting points. Now we must keep two (or more) front lists and two (or more) sets of records of the best place to have come from and the cost to get to each point. This is

Starting the flooding algorithm at three points

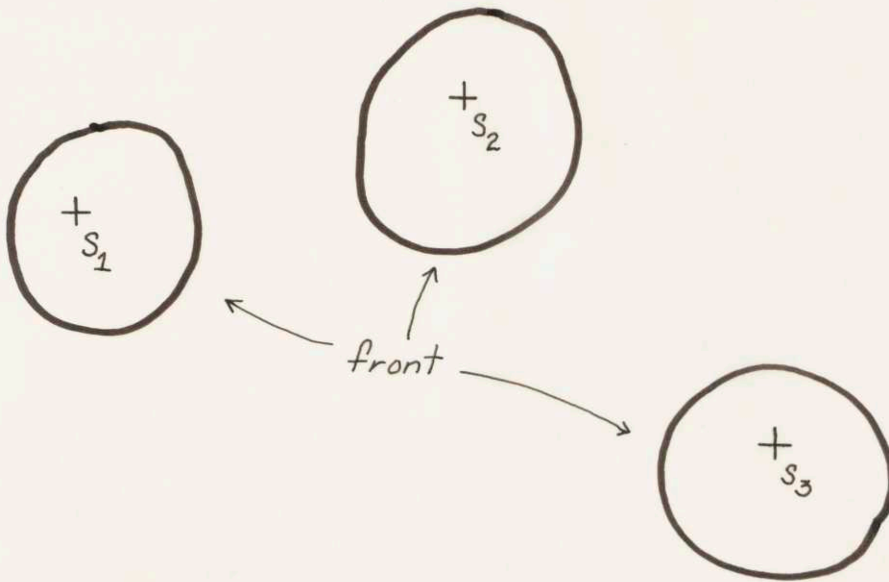


Figure 21

Fronts join after starting at three points

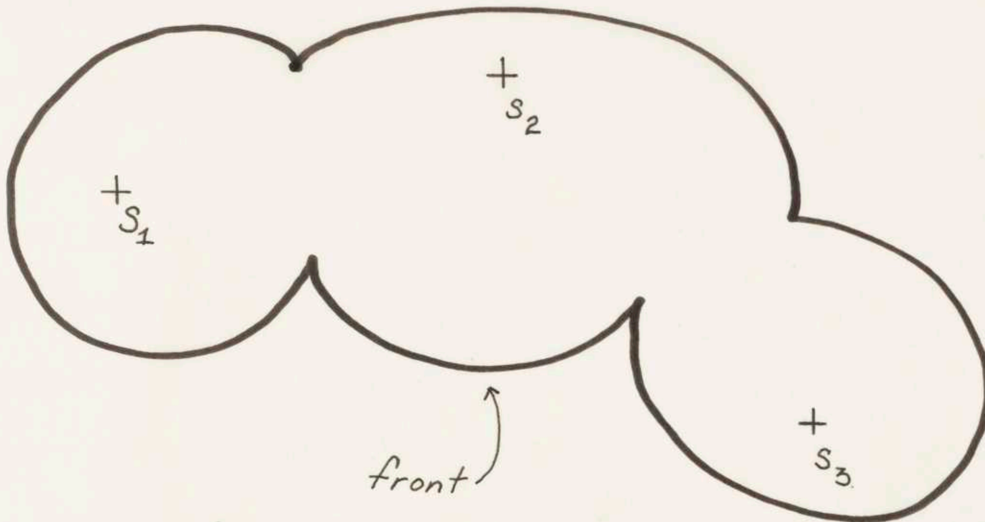
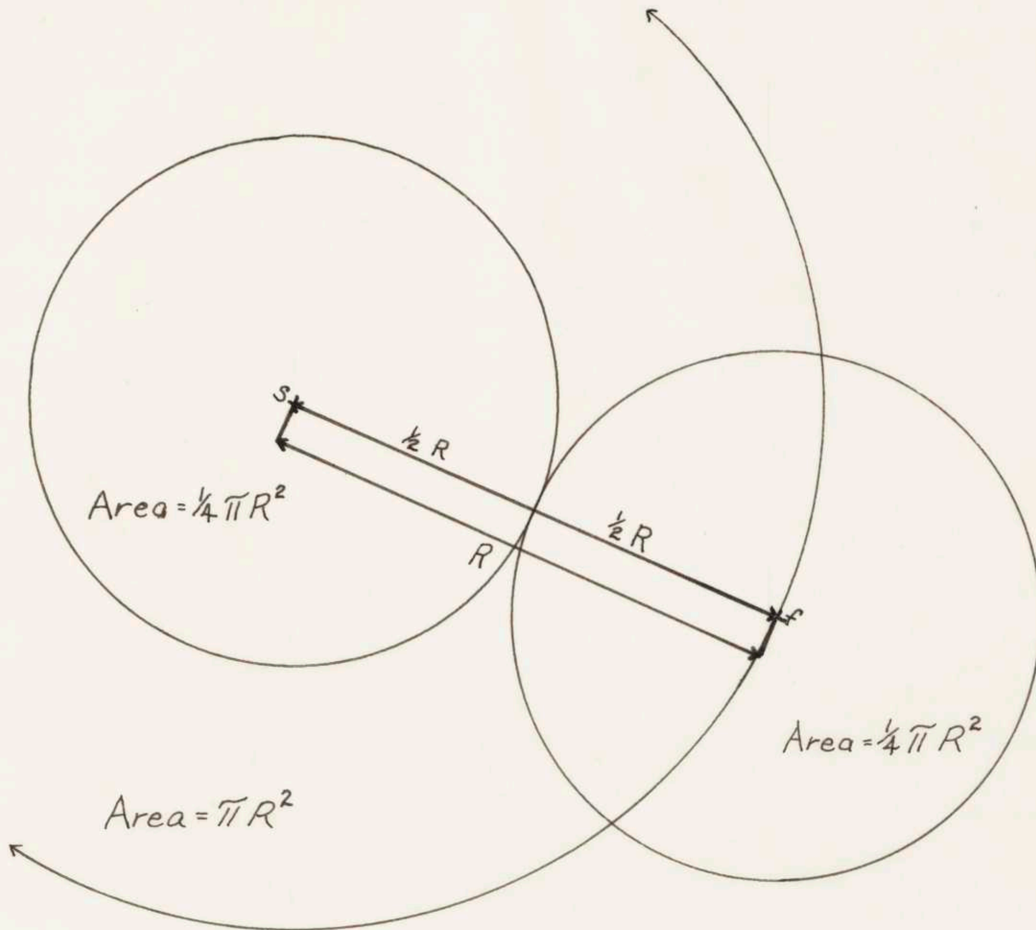


Figure 22

most easily accomplished by running the algorithm (finding paths) for each starting point and storing the paths and costs after each is found. Note that it is possible to implement any number and combination of AND and OR starting and termination conditions for the algorithm.

There is the possibility of making one-half as many calculations as when we let the algorithm proceed only from the path starting point, if we let the algorithm start from the path starting point and finishing point, and stop it after the two fronts cross. In an idealized case where we let the algorithm start only from the path starting point, the front will expand as a set of concentric circles. The number of points that have been expanded is proportional to the area of this circle. If the distance (in this case the number of points, not the cost) from the starting point to the termination point is R, then the number of points expanded is proportional to πR^2. If we now let the fronts expand <u>at the same rate</u> from the path starting point and the path finishing point, and terminate the algorithm when (or after) the fronts cross, the radius of each of the two circles is $\frac{1}{2}R$. The number of points expanded is then proportional to $2(\pi(\frac{1}{2}R)^2) = \frac{1}{2}\pi R^2$ (see figure 23).

Idealized case of starting a flooding algorithm
from two points



Area of large circle is twice the area of the
two smaller circles.

If the points in the space are of equal density,
there are one-half as many points inside the two smaller
circles as inside the larger circle.

Figure 23

As previously mentioned, the time to terminate the algorithm is when (or after) the fronts cross. There are two ways of deciding when to terminate the algorithm explained below. The first of these is perhaps the easier to implement. However, the time required for the algorithm to terminate for some uses may be longer than with the second. This is because the first method requires the termination decision to be made in the innermost loop of the algorithm, while the second method allows the termination decision to be made in an outer loop. We will now introduce the notation for the following discussion. There are two fronts, front A and front B. The points inside the fronts we will call a and b (points a are inside front A, points b are inside front B). See figure 24.

Termination Criterion for Algorithms Which Develop Paths From Both Ends Simultaneously

The first method is essentially the same as described by Nicholson.[20] The only differences are in notation and algorithm implementation (although the algorithms are implemented differently, they do the same thing). The criterion for termination of the algorithm is as follows. First, the cost to get to a point is computed. If this cost is less

Figure for discussion of termination criteria of the flooding algorithm

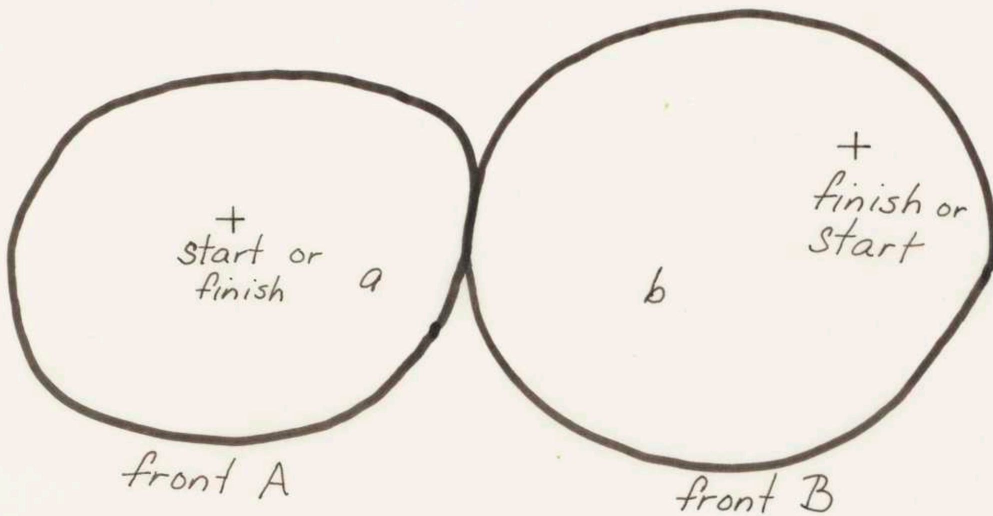


Figure 24

than any previously computed, the cost to get from this point to the other algorithm starting point must be checked (in addition to the other operations required by the flooding algorithm as described by the bottom section of figure 20-a). If this computation is finding paths from the start, then we must check the cost of getting from this point to the finish; and vice versa. If we find this cost to be less than infinity (or its approximation), then we have found a path from the start to the finish. If the cost of the complete path is lower than previously found, then replace the old total cost with this new cost. (Also replace the path.) The time to terminate the algorithm is when the cost of the cheapest path found is less than or equal to the cost of one front plus the cost of the other front plus the lowest cost transition possible.

One proof that this gives the lowest cost path is given by Nicholson.[20] A briefer proof is as follows. The position of the fronts will be as shown in figure 24. The lowest cost path from the path starting point to the finishing point can include as members only points that are <u>inside</u> front A or inside front B. If a point is outside front A and outside front B, then the cost to get to it from the start

will be greater than or equal the present cost of front B plus the cost of front A. Hence, the total cost of the path which includes this point as a member must be greater than or equal the sum of the present front costs. From above, the cost of the path we have found is less than or equal this sum, and there is no need to consider these possibilities. To continue the proof, we must investigate the properties of the fronts. Remember that it is impossible for a path to go from a point outside the front to a point inside the front without the path including a point on the present front. Therefore, to go from the start to the finish, we must cross both fronts; that is, any path must include as a member at least one point of each front. Each time a point is added to either front, the algorithm requires that we check the cost of the path from the start to finish that includes this point. It saves the lowest cost of these. Therefore, we have found the lowest cost path.

The second method of terminating the algorithm is as follows. Each time the top point is taken off the front list, a check is made to see if this point is inside the other front. If it is, the cost to get to this point is recorded as C_1. This has to be the lowest cost point on this

front that is also inside the other front. Let us call this front at this time front A', and likewise, the other front B'. Now we must search over the points on front B' to find which one has the lowest cost to get to the point from which this algorithm started. When this cost is found, it is recorded as D_1. Remember from the previous section that there is no need to consider points outside either front (front A' or front B'), and any path must contain at least one member of each of these fronts. Therefore, the lowest cost path that can be found is $C_1 + D_1$, the sum of the lowest cost components on the front.

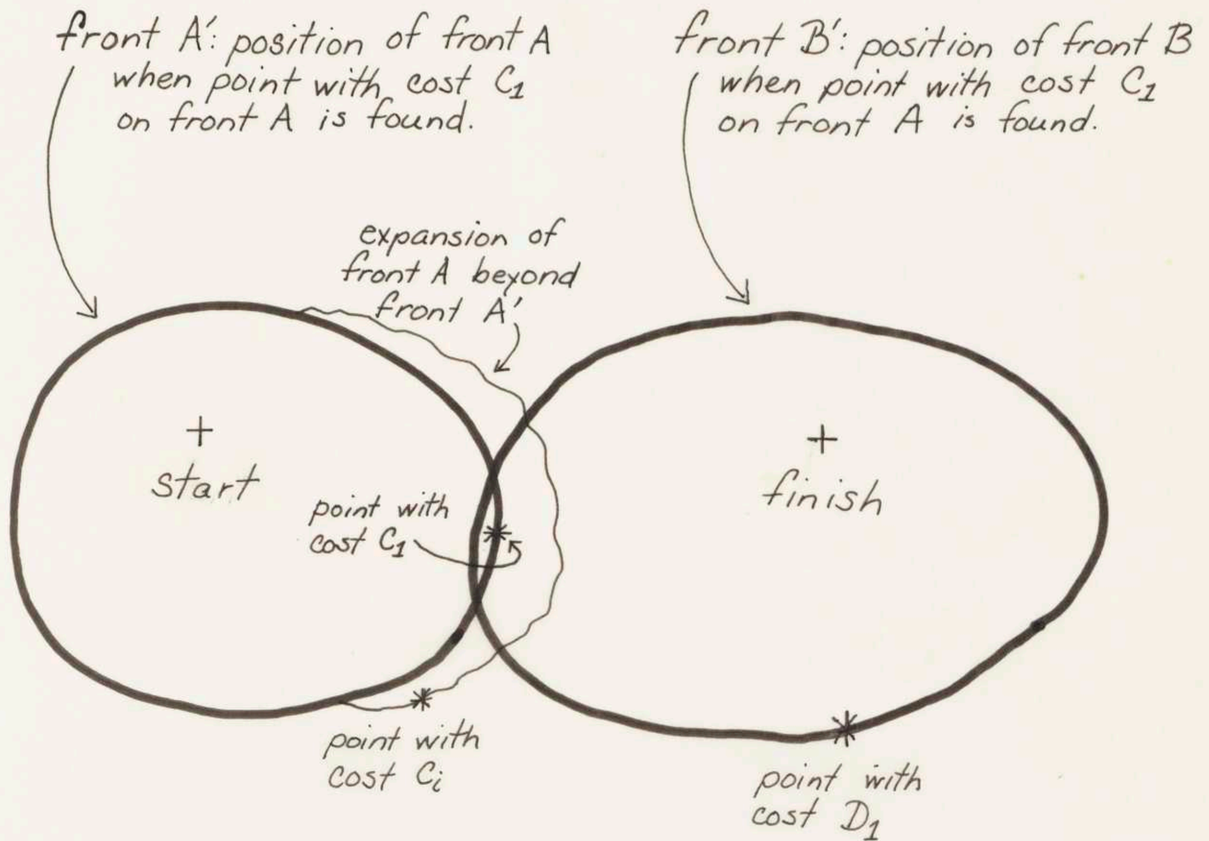
Now, we will not continue to expand the fronts at equal rates. We will continue to expand only front A (front A' is a particular position of front A). As each point is expanded, we must check for a lower cost path from the start to the finish, and replace the old cost with any new lower cost found (and the old path with the new path). We can terminate the algorithm only as soon as the cost of the point being expanded is greater than or equal to the cost of the lowest cost path yet found minus D_1. This procedure guarantees that the lowest path from the start to the finish has been found as any future path that will be found must have a cost greater than or equal to the present lowest cost path.

As we showed above, the lowest possible cost path is $C_1 + D_1$, and the lowest cost path must include at least one point that is a member of the set comprising front A' AND b (AND is logical and). When the cost of the point being expanded, C_i, plus D_1 is greater than or equal to the cost of the lowest cost path found, then we have investigated all points that are members of the set front A' AND b that could be on the lowest cost path. (C_i can never decrease; therefore, $C_i + D_1$ can only be greater than or equal to the present lowest cost path.) Figure 25 supplements the above explanation.

There are two trivial termination situations which deserve special consideration. Prior experience has shown that one or the other of these has always been the situation when the algorithm terminated. The first is when there is only one member of the set front A' AND b. Then the first path found is the only path possible. The second situation is when all the costs of the members of front A' AND b are the same. However, see figure 26 for a situation where this is not the case.

There is one further item to add. That is, when there is at least one member of the set front A' AND b, and there are no members of the set a AND b, (that is, front A' and front B' are adjacent where front A' is inside front B', or

Illustration to supplement explanation of Termination Criterion of flooding algorithm when paths are developed from both ends.



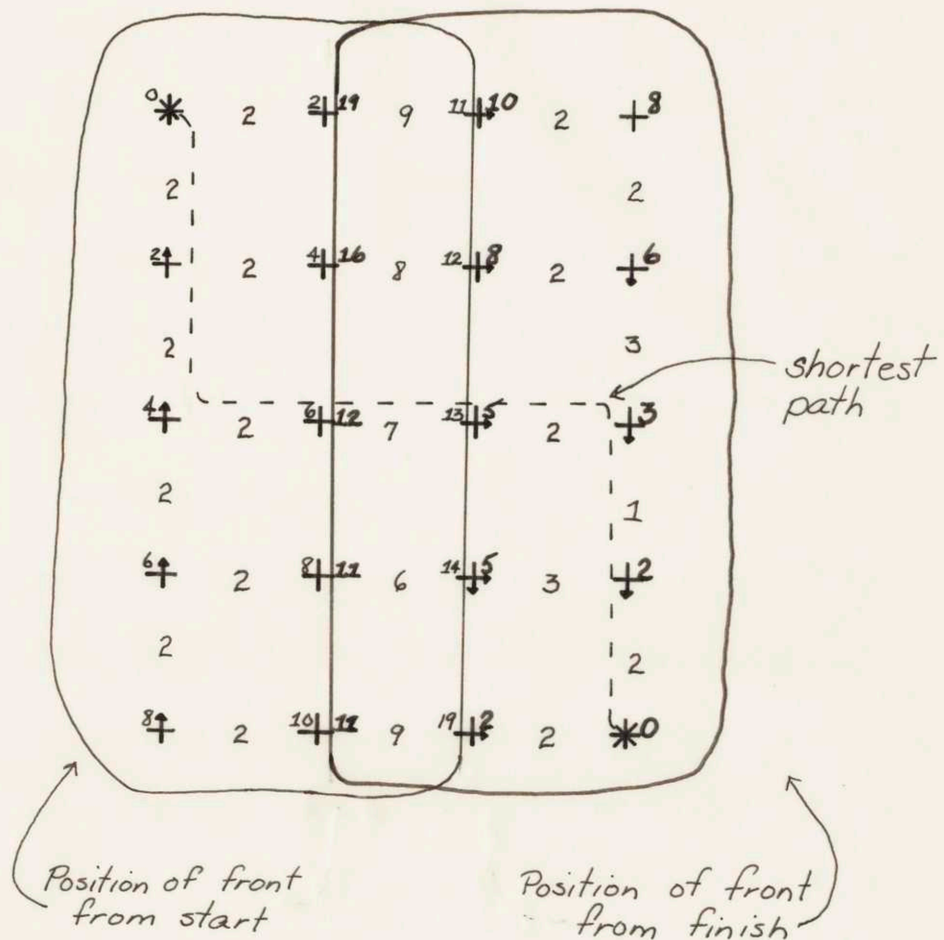
C_1 is cost of first lowest cost point on front A found to be inside front B′.

D_1 is cost of lowest cost point on front B′.

Front A is expanded from front A′ until C_i, the cost of front A, is greater than or equal to the cost of the cheapest path from start to finish minus D_1.

Figure 25

Situation in which complete inspection is necessary to find shortest path.



Numbers between points are transition costs. Missing costs can be taken to be infinity.

Light numbers at points (on the left) are costs to get from start to that point.

Boldface numbers at points (on the right) are costs to get from this point to the finish.

The cost to get to points where the fronts are adjacent and parallel are not equal. The lowest cost path is 18 cost units. It will not be the first path investigated.

Figure 26

vice versa), then the number of members of front A' AND b equals the number of members of front B' AND a. As the two fronts are adjacent, they must be parallel (not necessarily straight). As they cross one another at the ends of the parallel sections, the lengths of the fronts where they are parallel must be equal. And as the number of points on the fronts are proportional to the length of the fronts, then the number of points on each front in this region must be equal.

The A* Algorithm

The flooding algorithm just discussed is a special case of the A* algorithm.[11] However, having proved the flooding algorithm's properties we have a solid basis on which to proceed with the A* algorithm. There are no flow charts given for the A* algorithm as it is identical to the flooding algorithm except for expanding the point on the front (the open point) which minimizes $\hat{g}(\alpha)+\hat{h}(\alpha)$ (defined below) and the special cautions outlined below.

First a definition of terms. α is the point being discussed, any point in the space. $\hat{g}(\alpha)$ is the current best estimate of the lowest cost path from the start to α. $g(\alpha)$ is the cost of the lowest cost path from the start to α.

$\hat{g}(\alpha) \geq g(\alpha)$ in all cases. $\hat{h}(\alpha)$ is an estimate of the cost of the path from α to the finish. $\hat{h}(\alpha)$ must be greater than or equal to the cost of the lowest cost path from α to the finish. The set of terminal points for the algorithm is $\mathcal{T}$, and t represents a member of $\mathcal{T}$.

When implementing A*, it became apparent that special care has to be taken when there is more than one point in the terminal set, $\mathcal{T}$, and the terminal condition is to find the cheapest path to any one of the several members of $\mathcal{T}$. (The OR terminal condition.) Let $\mathcal{T}$ be composed of n members, $t_1, t_2, \ldots, t_i, \ldots t_n$. As $\hat{h}(\alpha)$ must be a function of α and t, $t \in \mathcal{T}$, we must, at every point α, use $\hat{h}(\alpha)$ where $\hat{h}(\alpha) = \min_i f(\alpha, t_i)$, to insure that the algorithm terminates properly.

When the terminal condition is AND (we must find the cheapest path to all members of $\mathcal{T}$), the same problem does not exist. One way of implementing this condition is to pick a t_i to compute $\hat{h}(\alpha) = f(\alpha, t_i)$, and continue with this t_i until it is closed. Then, scan the list of terminal points and pick one that is not yet closed, t_j, and use it to compute $\hat{h}(\alpha) = f(\alpha, t_j)$. Continue these steps until all points of the terminal set are closed.

The A* algorithm has the same properties as the flooding algorithm in the case of multiple starting points (OR) and starting the algorithm from two or more points (AND). Also, when the A* algorithm is started from the path starting and finishing points, the same termination procedures as described for the flooding algorithm apply (as $\hat{g}(\gamma)=g(\gamma)$ when γ is the open node that satisfies $\hat{g}(\gamma)+\hat{h}(\gamma) \leq \hat{g}(\alpha)+\hat{h}(\alpha)$ and α is any open node). However, in all cases in which there is more than one possible terminal state, the procedures described above must be used.

The Two Stack Diamond

One of the first objectives of this project was to find or invent shortest path algorithms that terminate quickly when finding paths through large numbers of nodes (of the order of 10,000 or so). One of the first algorithms tried was a variation of Minty's Algorithm.[23] In an effort to make this algorithm run faster, a list was kept of all the points whose cost (of the path from this point to the start) had decreased. A point was taken off the list when the costs of the paths to its neighbors were investigated (these paths being required to go through this point). This list of points composed a front.

In the course of some experimentation, it was found the best way to put points on the list and take them off was the First in-First out method. This method tended to make the front progress through the matrix of nodes perpendicular to the surface formed by the front. Using a Last in-First out method (a push-down list) tended to generate lines of nodes that started from the extremities of the front and were parallel to the surface formed by the front. This is a wasteful procedure as the paths formed along these lines would generally not be minimum cost and would have to be recalculated many times before a minimum cost path was found.

The only problem with the First in-First out method was that it required extra instructions for computer implementation as it could not use instructions for pushdown list manipulation. As an alternative, two stacks of front points were kept. Points whose cost had changed were put on one list, and points to expand were taken off the other. When the list the points were being taken from was empty, the lists were swapped. We now took points off the list we were adding them to, and vice versa. This procedure let the push-down list facilities be used, and moved the front perpendicular to the surface formed by the front.

A flow chart of this algorithm is shown in figure 27. The advantage of this algorithm is that very few decisions are made in the inner loop. When paths to many points, or widely separated points in the space must be found, then this algorithm is generally faster than the flooding algorithm or the A* algorithm. They both require many more decisions to be made in the inner loop. This greatly increases the time they require to terminate. Appendix A shows this algorithm finding the minimum cost paths in a space.

Flow Chart for the Two Stack Diamond Shortest Path Algorithm

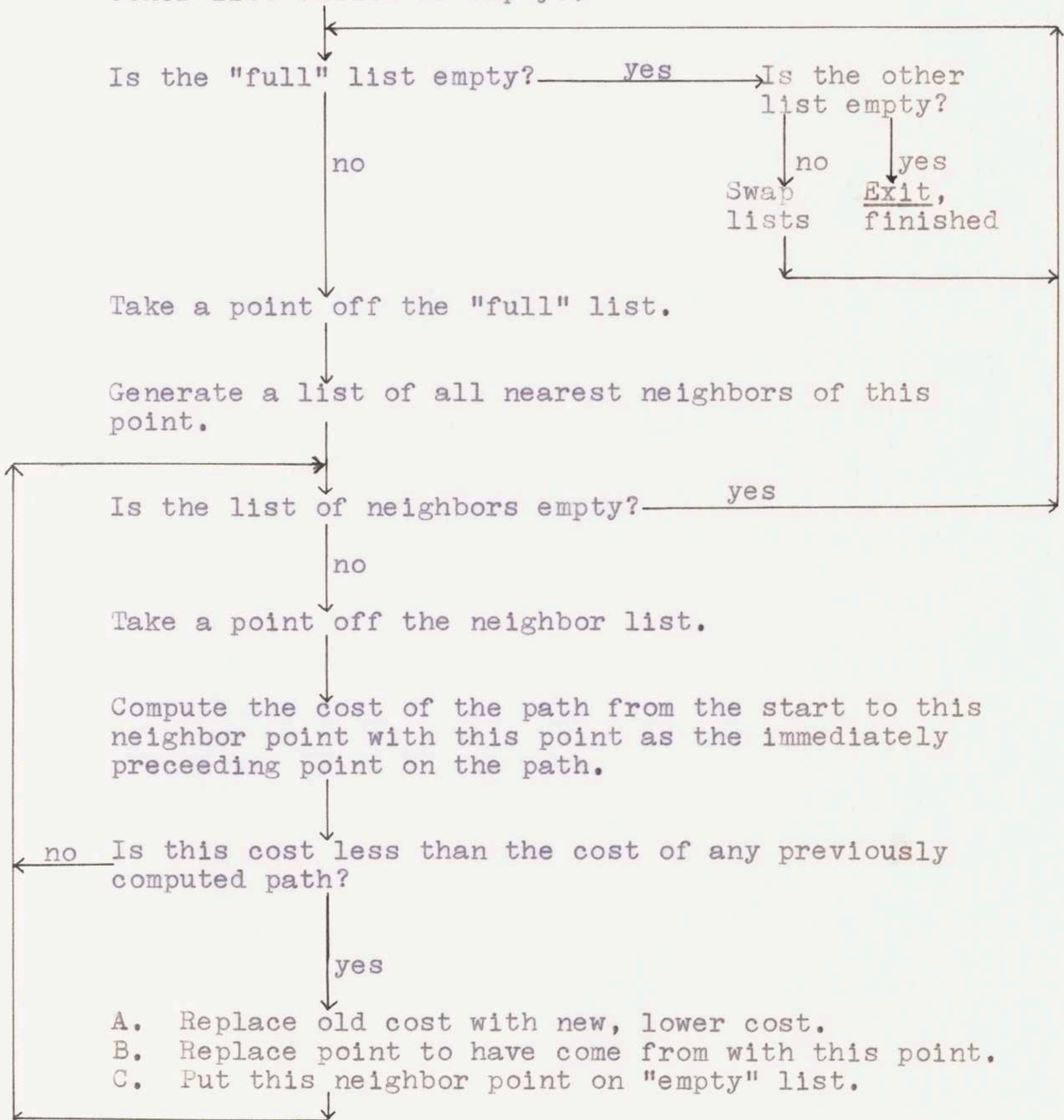


Figure 27

Chapter IV Implementation of Algorithms for the Purposes of This Work

The chapter begins by specifying the restrictions on the examples the demonstration system can solve and investigating some basic manipulation situations. In the body of the chapter, the methods used to implement the procedures discussed in previous chapters are explained.

Restrictions on the Examples and Demonstrations and Definitions

Two Dimensional Space

All tasks will be performed in a two dimensional space. All motion will be confined to be within the space, and the representation of all objects and the manipulator jaws will be two dimensional. This restriction is made to keep the size of data storage needed for the state space model of the task reasonable. The principles on which the system is designed are extendable to three dimensions (or more), but of course, the data storage and the processing time increase according to the number of dimensions required by the state space model to describe the task.

No Rotations

For the same reason, the system will not allow any rotations. Rotations of objects can be completely described by a model of sufficient dimensionality. Not allowing rotations keeps the number of dimensions in the model small. As before, the system can be extended to include rotations if necessary.

Digitized Object Shapes

The objects that the demonstration system can represent can be of any two dimensional shape, but the sides must be parallel to the X or Y axes of the space, and the object must be connected. Figure 28 is a representation of an object that can be represented on the system.

Movable and Immovable Objects

The system will divide objects into two classes, movable and immovable. In our everyday experience, we make this classification, but the attribute immovable has meaning only in a relative sense. As far as is known, there is no absolutely immovable object. When we are thinking of moving furniture, we consider the walls of the house as immovable. But wreckers move (or remove) walls or entire buildings. Likewise, there are other groups that

Object that can be represented on the demonstration system

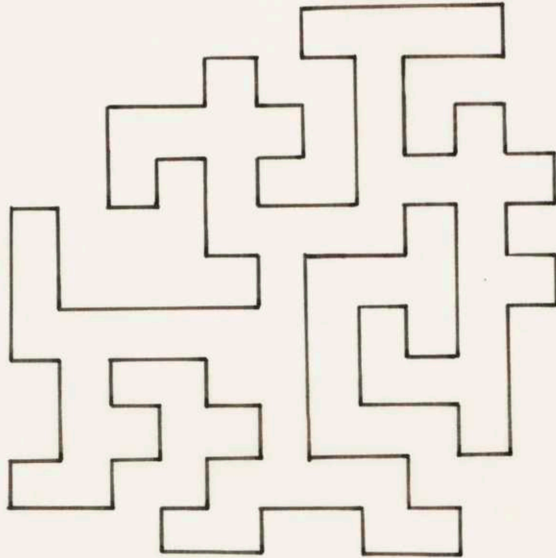


Figure 28

specialize in moving things we normally think of as immovable. In the examples and demonstration, objects are defined as movable or fixed. The system will never move a fixed object.

Only One Object at a Time Can be Moved

This demonstration system cannot move more than one object at a time. This means, for example, that the system cannot use one object to push others, nor can it use an object as a tool in an active manner.

Description of Manipulator Jaws

The things that move the objects are the representations of manipulator jaws. These are an idealization of an end on view of a pair of no-slip jaws of a three degree of freedom manipulator (X, Y, open-close). The jaws will normally move from place to place closed, but may move open. Also, they will normally grasp objects to move them, but they can also push. Figure 29 shows the jaws in various states.

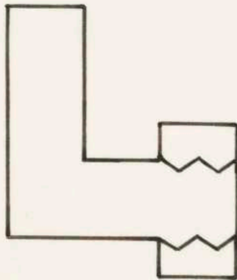
Definition of Temporary Location

The characteristic that identifies a Temporary Location is that if an object is put in a Temporary Location, then the jaws can move, unimpeded, completely around the

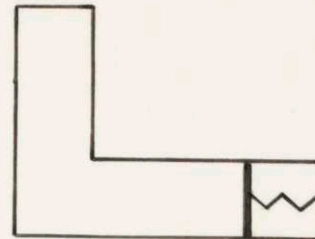
Manipulator Jaws

a) jaws closed

b) jaws open



c) jaws grasping
an object



d) jaws pushing
an object

Figure 29

object. Note that what is a Temporary Location for one object is not necessarily a Temporary Location for another object. The value of Temporary Locations is that if an object is moved to one, then the object can be moved from it to another location. This follows directly from the definition. One deficiency of the demonstration system is that many times it would be satisfactory to move an object to a position (not necessarily a Temporary Location) from which it can be moved. But the system has no way of deciding which locations in the space satisfy this requirement. So the system moves the object to a Temporary Location. There will be a discussion of this and associated problems later.

Definition of Out of the Way Place

The next term is Out of the Way Place. This is a set of locations in the task space which are Temporary Locations, and in addition, are out of the way of all paths the system has planned, but not yet executed. How the system determines which locations are Out of the Way Places will be explained later in this chapter.

Definition of a Simple Task

For our purposes, a simple task is defined as moving one object. This task must be able to be executed without

any restrictions or qualifications. Operationally, this is equivalent to saying that a shortest path algorithm, designed to find the path for one object, is capable of finding the solution to the simple task. Figure 30 is an example of a specification of a simple task.

As well as specifying the task pictorially, we can also specify it using the following words:

MOVE A TO LOCATION 20,20.

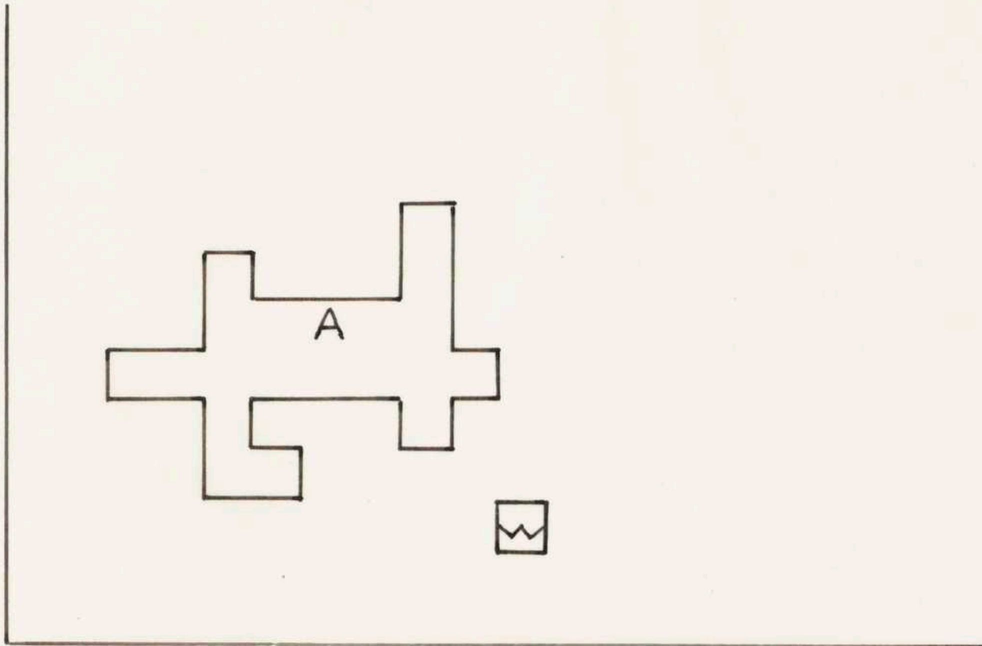
A more concise way of saying the same thing, given that we are talking about moving objects to places that can be specified by numbered coordinates, is to give only the name of the object and the location to which it is to be moved, specifically:

A, 20,20.

In this restricted case, the character string, <name, location> completely specifies the task. We can call this character string an abstract specification of a task. Such a character string does not include all of the information about a task, but does include enough so that a computer system (Whitney's, for example), or a person, can determine the remaining information necessary to execute the task.

Specification of a Simple Task

Initial Positions



Final Positions

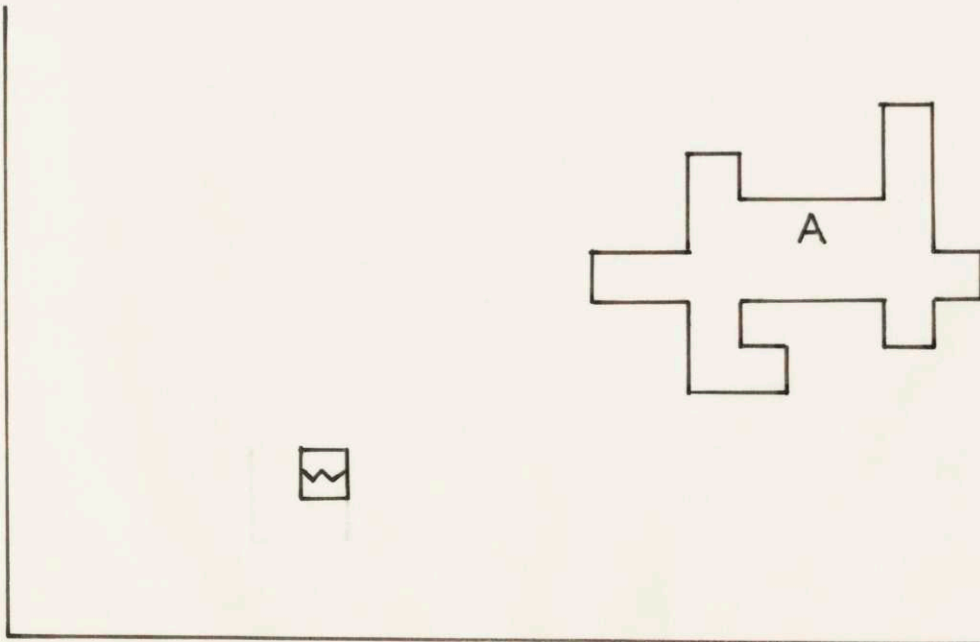


Figure 30

Some Basic Manipulation Situations

In conjunction with the restrictions given on the previous pages, there developed the necessity to study some basic manipulation situations. As the reader will see, they are all more or less straightforward, but it was felt they deserved special consideration.

Situation 1

Even if an object has the attribute of being movable, it may not be possible to move it. Its position, or the objects that surround it, make it impossible to move. In the system, if an object is described as movable, but is surrounded by immovable objects, the system has to discover that the object cannot be moved. It does not know the object is immovable without first attempting to move it and discovering in the internal model that it cannot be moved.

An example of such a situation is shown in figure 31. The movable object, A, has only two sides covered by the immovable object, but as A presents no handles for grasping, or surfaces on which to push away from the immovable object, A cannot be moved. There are many more similar situations. The system discovers that an object is, in effect, immovable by attempting to move the object and finding out (by the

Movable two dimensional object cannot be moved out of corner

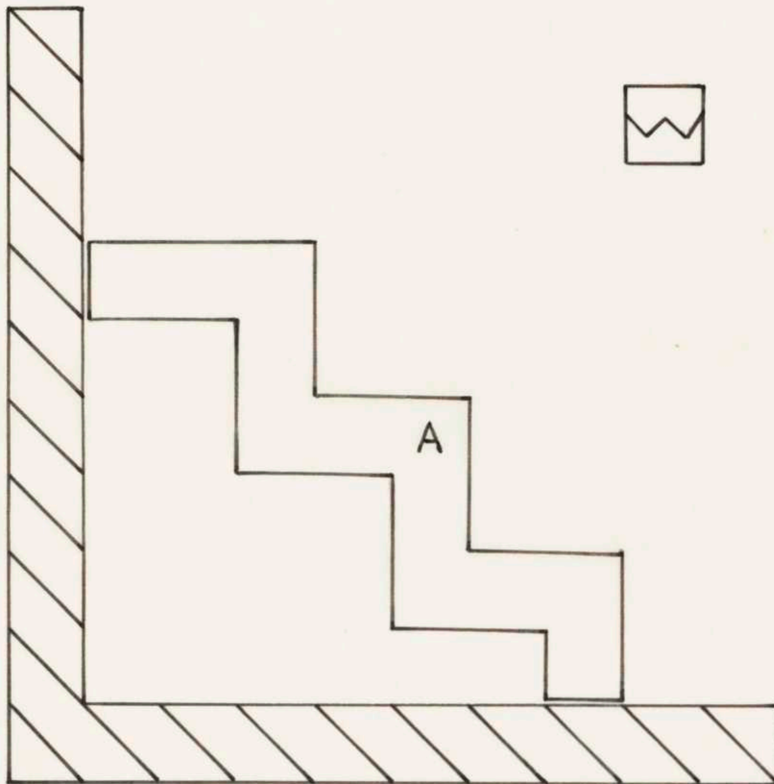


Figure 31

failure of the shortest path algorithm to find a solution) that the object cannot be moved.

In conjunction with the above, let us point out that an object can be moved into a position that it cannot be moved from. For example, if in figure 31, object A were a few spaces to the right, it could be pushed to the left into the corner, where it would again be immovable.

Situation 2

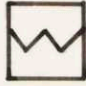
Given that an object is in a position it can be moved from, then it is not always possible to move the object to a desired position in the space. Usually, this situation occurs because the desired position is blocked by one or more immovable objects. Figure 32 is an example of such a situation.

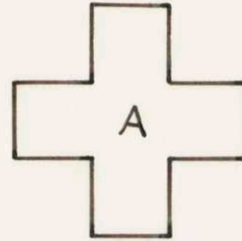
The demonstration system will discover this situation only if it attempts to move an object to such a position. If this situation is discovered, the task being executed is declared to be impossible.

Situation 3

Given two positions, P1 and P2, that objects can be moved from, and given that an object can be moved from P1

Object A cannot be moved to position X in two dimensions





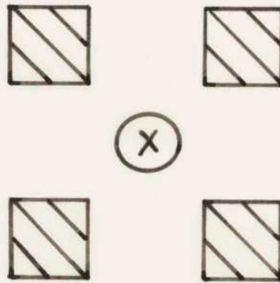


Figure 32

to P2, then it is not always possible to move the object from P2 to P1.

This states that diodes exist for objects, that an object may be moved in one direction through a narrow opening between immovable objects but may not be able to be moved in the other direction. In figure 33, the movable object A can be moved by the jaws from P1 to P2, but the jaws cannot move it back. There are many more possible situations where diodes are encountered.

In the remainder of this work the assumption will be made that diodes do not exist in the task space. If a diode is present in an example, the demonstration system may make a fatal mistake in attempting to solve the task.

Situation 4

Before an object can be moved, that which is to cause motion must be in contact with the object. The basis for this statement is that an object must be acted on by a force before it can move. Of course, there are situations where physical contact is not necessary to move an object. In these cases, some other force such as magnetic or electrostatic must be used. But in the demonstration

A can be moved from P1 to P2, but not from P2 to P1

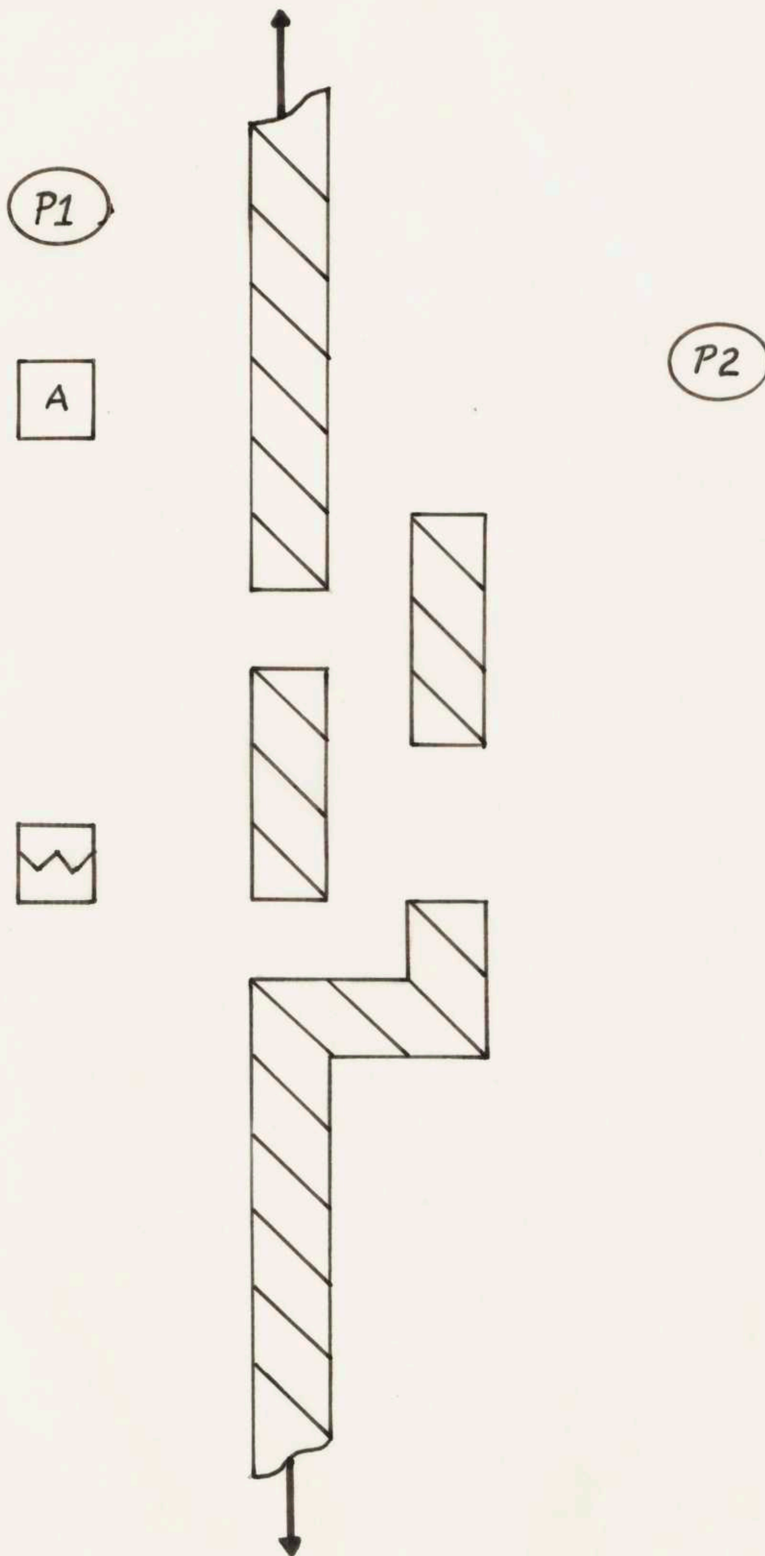


Figure 33

and examples only the jaws, acting directly, can move objects.

Generation of a State Space for an Object of Any Shape

As stated previously, a state space can describe a manipulation task, and a shortest path algorithm can find the solution of a manipulation task by finding a path through the state space. The state space method is used to find solutions to simple tasks, and the solutions of several simple tasks are concatenated to form the solution of a complex manipulation task. The TASK TREE orders the simple tasks so their combined solutions form the solution to the complex task. This is described in this and the following chapters.

The first step toward finding the solution of a simple manipulation task is the generation of a state space. To generate the state space, the system first needs a description of the physical space and of the object to be moved.

These descriptions depend in some part on the exactness with which they are made. The exactness or precision with which the space is described will be called the "quantization" or "reticulation" of the space. If the quantization is very fine, many surface irregularities of the objects in the space

will be included in the description. All this information probably is not necessary. On the other hand, if the scale of quantization is coarse, not enough information about objects' shapes will be known for good system performance. An object could not be grasped or pushed reliably, for example. The scale of quantization is probably best determined after a task has been specified. Accordingly, this system has no specified quantization scale. (However, the quantization scale for the demonstration is constant.)

For this system, the description of the space is an input. In other systems, it may be desirable for the system to be able to provide itself with the description of the task site. The input would be some lower level (less processed), more readily available information from the space. For example, it might be desirable to provide the system with the ability to "see." See, for example, "Recognizing Convex Blobs," by J. Sklansky,[29] or the sections on Artificial intelligence in M.I.T. Project M.A.C. Progress Report VI.[17]

Alternatively, it may be desirable for the system to be able to understand a Natural Language description of the space. See the Simmonds[28] article for numerous references.

In the present system, a space is described in terms of X-Y coordinates. An object's position is identified by reference to its "base location," a location on an object that is specifically designated. The base location occupies a square on the object, one quantization unit on a side. As most objects will be larger than this, the remainder of the object is described by specifying the other locations the object occupies. Figure 34-a is the drawing of an object. Figure 34-b is the list of locations the object occupies and the specified base location. Figure 34-c is the computer interpretation of the object's shape. The computer description lacks much of the information given by the drawing of the object. But it does give the system something to work with, and the computer description of objects is not a goal of this thesis.

We want to generate a state space to describe the motion of an object in a physical space. The first step is to get a computer description of the physical space, excluding the object to be moved. The next step is to use the description of the object to be moved. Then the object's base location is put at every location in the physical space, every node or state in the space. If any part of the object

Digitizing an object's shape

Object in a digitized space

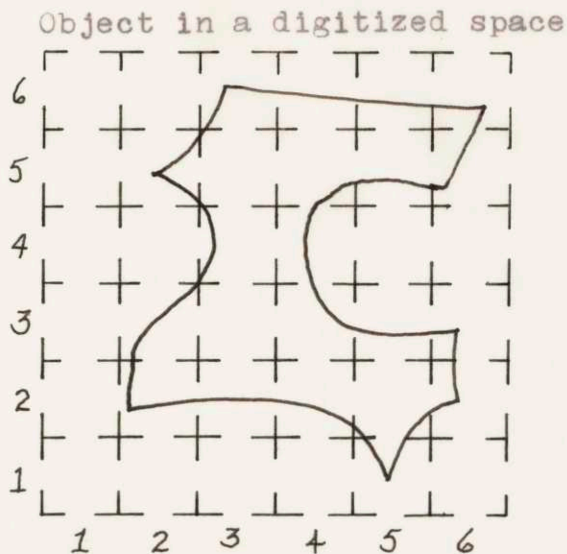


Figure 34-a

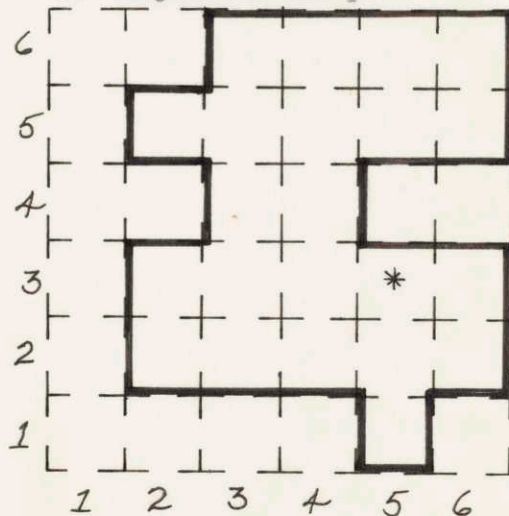
Base Location 5,3

Locations occupied:

2,2	4,2	5,3
2,3	4,3	5,5
2,5	4,4	5,6
3,2	4,5	6,2
3,3	4,6	6,3
3,4	5,1	6,5
3,5	5,2	6,6
3,6		

Figure 34-b

Computer interpretation of object's shape



* is at base location

Figure 34-c

is on an immovable object, or out of the space, a flag is set that says the object cannot occupy this state. This state becomes a forbidden node. If the object is "on" a movable object, this is indicated by calculating an extra cost for moving to this node from each of its neighbors. This "extra cost" is used to make the system behave in a "desirable" way. An example of "desirable" behavior is "it is better to go around an object than to move through the space occupied by it." The question is, how far is one willing to go to avoid moving through an object? The answer depends on how difficult the object is to move. For example, most people will not walk more than a few feet out of their way to avoid moving a light chair. But most people would rather walk a moderate distance than have to move a large table. This "extra cost" is made proportional to the size of the object.

The reader may be disturbed that the system can move one object through the space occupied by another. Actually, the system will not do this. What it can do is to _plan_ to move one object through the space another object occupies. This is analogous to the way one might plan to move a large piece of furniture out of the house. First, the best path out of the house is estimated, and a list of the

chairs, tables, etc., that need to be moved out of the way is made. These are moved out of the way, and then the large piece of furniture is moved. The furniture mover plans to move one piece of furniture through the space occupied by others, the same way the system does.

The flag for the object being "on" a fixed object and the extra cost information are all the information needed from the physical space. As described, the state space is a mapping of physical space. However, there must be more information in the state space to describe how the object is to be moved. We need an added dimension in the state space, which has five values: one to denote that the object is being grasped, and one for each of the four push directions ($\pm X$, $\pm Y$). This is the minimum set needed to describe motion of the object. The axis for this dimension is named the G axis. The complete state space for moving an object consists of the set of nodes $\{x,y,g\}$ which specifies an object's position in X - Y space, and how the jaws are in contact with the object.

The state space that describes the jaw's motion is the same as an object's state space, except for one difference. The jaw's state space does not need a dimension to indicate how they are moved. But the jaw's state space does

need a dimension to describe their opening and closing. A node in the jaw's state space defines values for the X coordinate, the Y coordinate, and how far open the jaws are, the K coordinate. The jaw's state space is generated in the same way as an object's state space.

The state space that describes the jaw's motion is used when the jaws are not in contact with the object. This state space is used to find the path the jaws take to get to the object initially, to change from a grasp position to a push position, or to change push positions.

Moving an Object from its Initial to a Final Position

The A* Algorithm discussed in the previous section finds the path for the object. The implementation procedure is straightforward and simple, but because of the way push and grasp positions are recorded, there is a possibility of trapping the jaws. Figure 35 is an example of a task in which the jaws get trapped moving an object. The system's solution for this problem, as the system has been described to this point, would be for the jaws to grasp the right "arm" of object A and move it to the right to its final position. The jaws would be trapped, and could not return to their specified final position; the task as specified could not be completed.

Specification of a task in which jaws could be trapped

Initial Positions

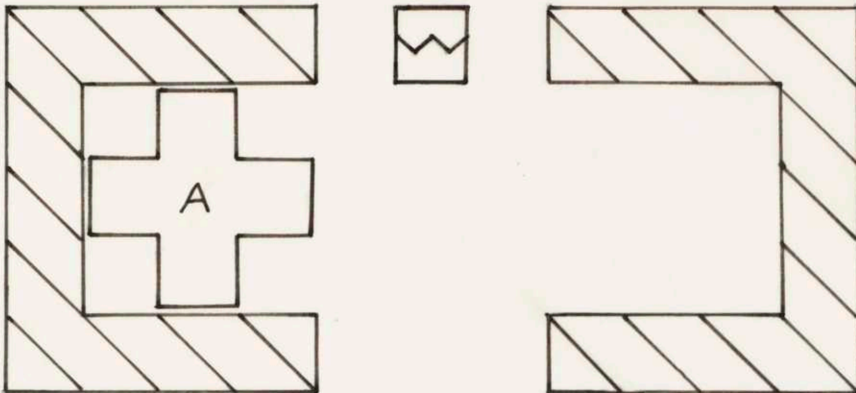


Figure 35-a

Final Positions

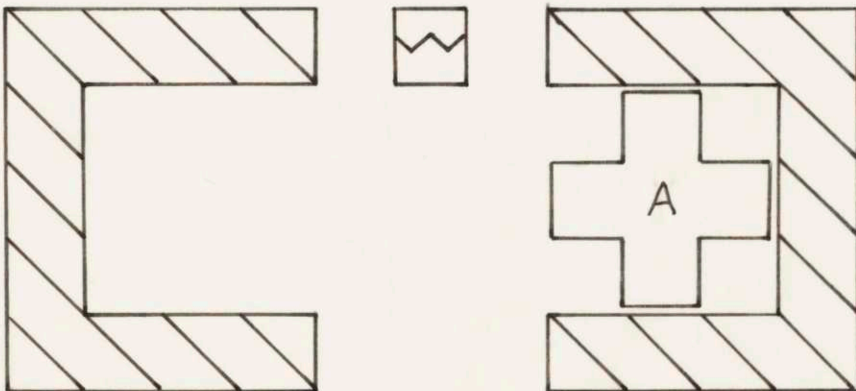


Figure 35-b

To prevent the jaws from getting trapped, two shortest path algorithms are used, one starting from the object's initial position going toward its final position, the other starting from the object's final position and going toward its initial position. The termination criteria are

1) that a path has been found from this point (any point in the space which the shortest path algorithm is currently considering expanding) to the object's initial position and to the object's final position; and
2) that with the object at this point, the jaws can get from their position as defined in one shortest path algorithm to their position as defined in the other shortest path algorithm.

In the previous section, conditions were given for terminating a set of shortest path algorithms. These conditions are purely mathematical and do not allow for the above problem.

Because of constraint 2 from above, the overall path may not be optimal; but, the path that is found will be near-optimal, and each of the two segments will be optimal. In general, this is good enough, as a near-optimal path will get the object moved in a reasonable manner.

In all cases, though, it is not necessary to have

the two shortest path algorithms running at each other. In tasks where the jaws cannot get trapped, only one algorithm may be used. Unfortunately, it is difficult to determine whether the jaws will be trapped when moving an object to a specified position. Hence, a test that is easier to implement, but is more restrictive, is used. The test determines if either the object's initial position or its final position is a Temporary Location. If either is, the system starts one algorithm that runs toward the position that is the Temporary Location. If both positions are Temporary Locations, the system starts the algorithm at the initial position. If neither position is a Temporary Location, the system starts the algorithm from both ends.

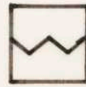
When the system moves an object to an Out of the Way Place, a different version of the A* Algorithm is used. First, the final location is guaranteed to be a Temporary Location; so only one algorithm is used. It starts from the object's starting position. Second, the system has no way of knowing *a priori* the best Out of the Way Place. Hence, $\hat{h}(\alpha)$ cannot be calculated as the terminal point, t, is not known. So a modified A* Algorithm is used, which sets $\hat{h}(\alpha)=0$ for all α. This is the Flooding Algorithm described in the previous chapter. This method finds the Out of the Way Place that is the cheapest to move the object to.

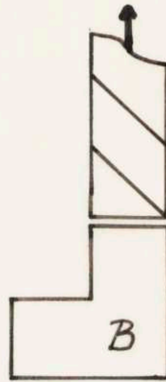
Discovering Which Objects Are in the Way

In some cases, the shortest path algorithm plans a path that requires moving an object through the space occupied by one or more other objects. In these cases, the object is not moved, but a list is made of those objects which are in the planned path, and tasks are generated to move these objects out of the way. Planning to move one object through the space occupied by another is the way the system discovers which objects must be moved out of the way.

The Out of the Way Place for an object is found in the following way. Say the system is given a task as described by the Initial and Final positions of figure 36. The aspect of this task to be noticed is that the system has to move B to an Out of the Way Place. The system goes through the following steps. It attempts to move A to its final position. It checks over the path it planned for moving A and discovers that the planned path includes space occupied by B, and that B must be moved so that it will not be in A's path. The system must move B to an Out of the Way Place. The crucial point of finding an Out of the Way Place is remembering the planned path, i.e., all the locations over which A is moved, including its initial and final positions.

Task in which B must be moved out of A's planned path





Initial Positions

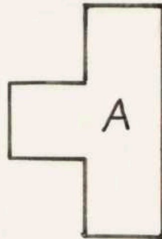
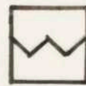
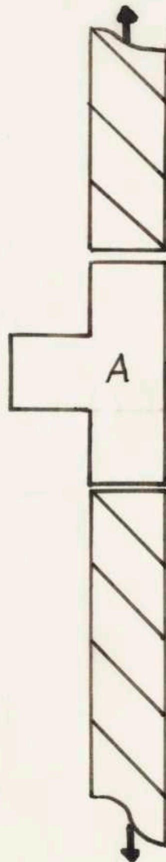


Figure 36-a





Final Positions

Figure 36-b

The system remembers this path, all the locations, as a dummy object A'. The system then moves B to an Out of the Way Place, a Temporary Location given the dummy object A'. (The system does allow movement over A', but not A, without worrying that A' ought to be moved before B is moved.) In some cases there may be a great number of paths that have been planned, and an equivalent number of dummy objects. An Out of the Way Place in these cases would be a Temporary Location, given all the dummy objects.

The rule that an object must be moved to a Temporary Location as an Out of the Way Place is overly restrictive. To move the object to a location where it is no longer on the space occupied by any of the dummy objects, in the large majority of cases, would be satisfactory. But if this rule is used, there will be cases in which an object is moved to a position from which it cannot be removed, and later in the task, the need may arise. Requiring that the object be moved to a Temporary Location guarantees that it can be moved later if necessary.

Finding the Order in Which to Move Several Objects to Final Positions

The system, as described, works very well unpiling a

stack of objects. But as so far described, it has no way to put more than one object at a specified final position. To overcome this deficiency, the ability of the system to unpile stacks of objects will be used to find the order to put them into a specified pile. To accomplish this, the task is reversed; that is, the time sense of the task is reversed and it runs from finish to start. The objects with specified final positions are moved from their final positions to their initial positions. Objects that have no designated final positions remain at their initial positions. (See Appendix B for the solution of a problem this method poses.) The TASK TREE as previously described is used to keep track of the order the objects are moved. As there are no gravity effects, springs, clips, etc., and the time sense of the task is reversed, the order that objects are taken out of the pile is the reverse of the order that they would have to be put into the pile. The system remembers the order the objects are moved <u>from</u> final positions. Once an object is moved from its final position, it can be moved any place where it will not interfere with moving the other objects. The easiest thing to do is to make the object disappear from the system's internal model of the space.

The system sets up the TASK TREE for moving objects to their final positions by putting the tasks on the TASK TREE in a single stack in the reverse of the order found above. Then, as each object is moved to its final position, it is set as immovable to insure that the system won't move it out of the way of some other object.

After each object is moved to its final position, a check is made to determine if any other object with a specified final position has been moved. If one has, then the order in which the remaining objects are moved to their final positions must be recalculated. The reason for this is as follows. Suppose that when the system first determines the order to move objects to their final positions, several objects' initial positions are also their final positions. The system cannot determine the order to move these objects as it doesn't have to move them. Later, during the task, one or more of these objects is moved out of the way. Now the objects' final positions and present positions (their present initial positions) are different, and the system must determine an order for moving them to their final positions.

In summary, the part of the system that determines the order of moving several objects to final positions works as follows. The system determines an order, puts this on the

TASK TREE, then moves objects until the first object is moved to its final position. Then the system checks if any other objects with specified final positions were moved. If they were, the system determines an order for the remaining objects, and continues. In a worst case, after moving each object to its final position, the system would have to recompute the order for moving the remaining objects to their final positions. But as there can be only a finite number of objects in a finite space, the system will complete the task.

Chapter V Capability of This System to Find Solutions

In this chapter, types of tasks for which this system will find solutions will be discussed. For the discussion, restrictions previously explained will be assumed to hold. We begin by investigating those tasks for which the system might not find solutions.

Tasks for Which the System is Not Assured of Finding Solutions

Before beginning the explanation, Connected Out of the Way Places must be defined. To aid in the definition, a simple example will be used. Say the system is planning to move object A, and finds that object B must first be moved out of the way. B is moved to Out of the Way Place X1. In the space there is another Out of the Way Place X2. X1 and X2 are Connected only if B can be moved from X1 to X2 after A is moved to its final position, without having to move A. In general, two Out of the Way Places, X1 and X2, are Connected only if an object can be moved from one to the other without having to move any objects that are in their final positions. Note that what might be Connected Out of the Way Places for one object might not be Connected Out of the Way Places for another object.

Two contrasting examples follow. Figure 37-a shows an initial position and figure 37-b the final position. In this task, all Out of the Way Places in the space are connected.

In figure 38, objects Y and Z are fixed. Figure 38-a shows the Initial Positions and figure 38-b the Final Position. In this task, Out of the Way Places X1 and X2 are not connected. B cannot be moved from one to the other after A is moved to its final position. The situation shown in figure 38 leads to a task the system cannot solve. The initial position could be as shown in figure 38-a, with the final position shown in figure 38-c.

The solution to this task is to move B to the left, then A to the right, then move B to an Out of the Way Place on the right side of Y and Z. Then move A to its final position, then B to its final position. The key to solving this task is moving B from the left side of Y and Z to the right side. This step moves B to an Out of the Way Place from which the task can be solved. The left and right sides of Y and Z will not be Connected Out of the Way Places after A is put into its final position. A solution is possible only if B is in the set of Connected Out of the Way Places to the right of Y and Z.

Task in which all Out of the Way Places are connected

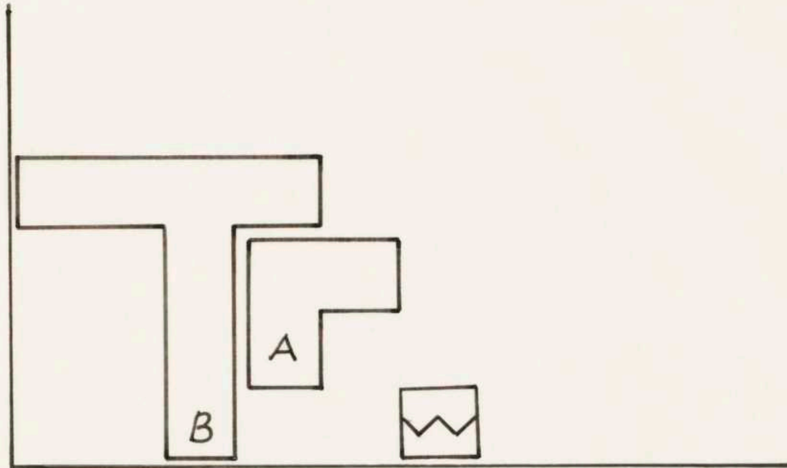


Figure 37-a

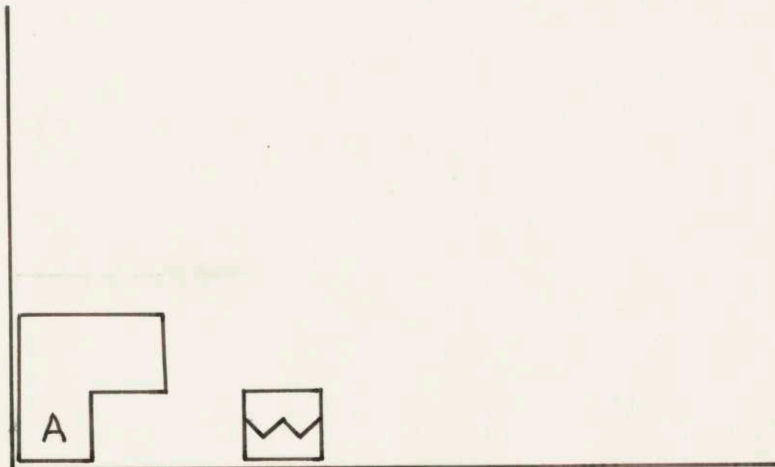
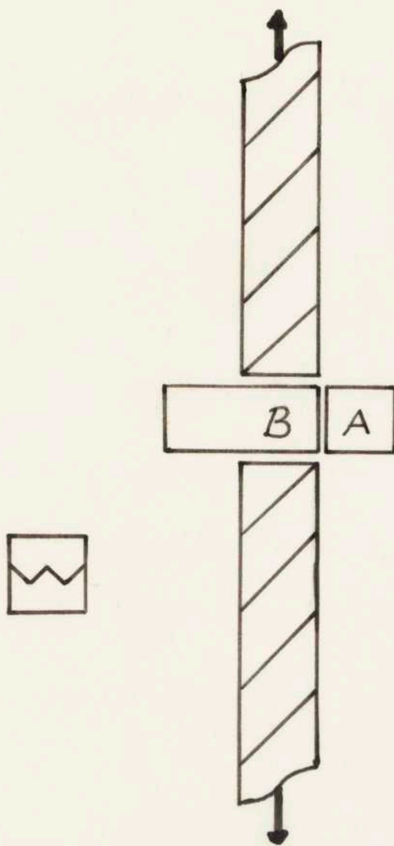


Figure 37-b



Task in which all Out of the Way Places are not connected

Initial Positions

Figure 38-a

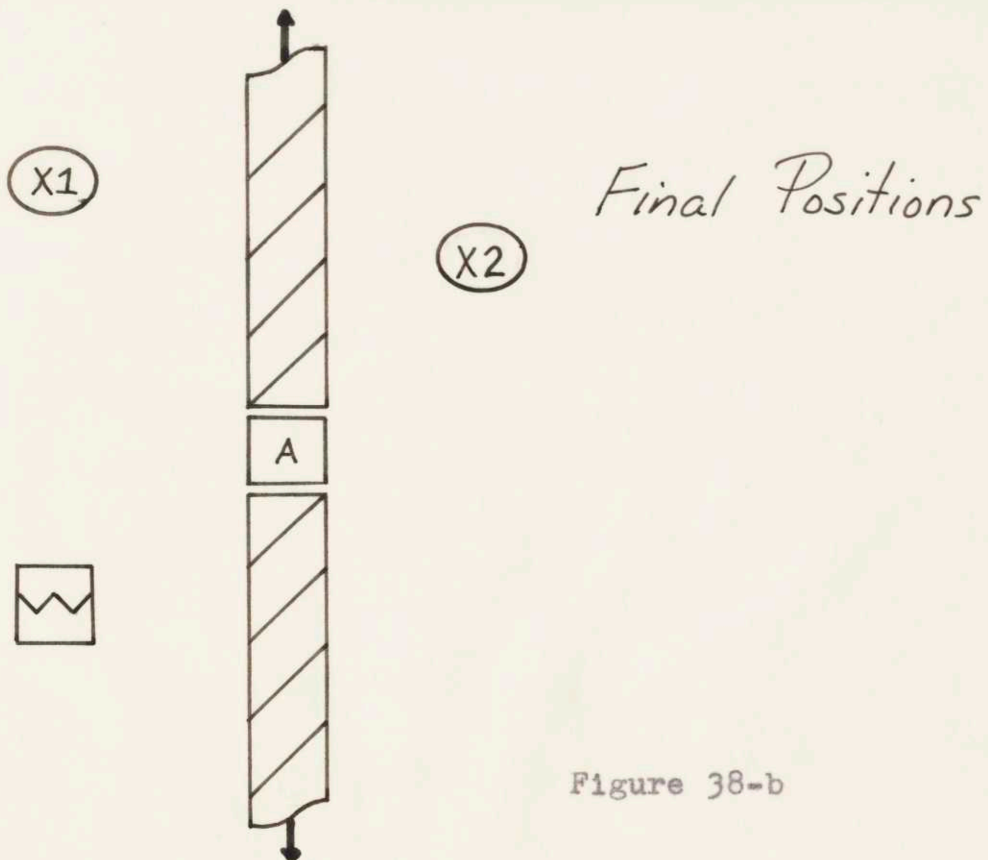


Figure 38-b

Final Positions of a task that the demonstration system cannot solve

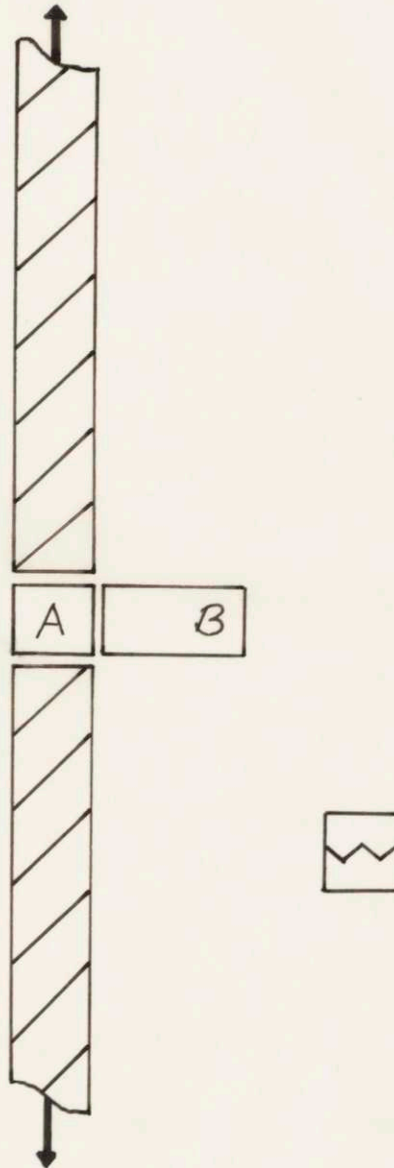


Figure 38-c

The system would attempt to solve the problem as follows. B would be moved to the right, then A would be moved to its final position and set immovable. The system is now blocked. It will not be able to move B to its final position. Note, if A were not set immovable the system would be stuck in an endless loop, moving one object out of the way, the second to its final position, then Out of the Way again, then the first to its final position, and so on.

What the system needs to solve problems of this type is a routine that divides Out of the Way Places into Connected sets, and determines which of the sets of Connected Out of the Way Places are acceptable for solving the task.

It is possible for the system to solve tasks that have non-Connected Out of the Way Places. But the system can fail if the task is such that the system doesn't happen to find the solution.

In cases where the system will not find solutions, the operator can give the system a sequence of intermediate final positions before he gives it the desired final positions. This method guides the system toward the solution. In the case of the task specified by figures 38-a and 38-c, the operator

could give an intermediate final position for A and B to the right of Y and Z.

There is one other assumption the system makes. It assumes that the jaws can get to their final positions. The system is designed to make sure the jaws are not trapped after they move an object, but it is not designed to make sure the jaws can get to their final position. The only way the jaws cannot get to their final position is for the jaw's final position to be blocked by a movable object which has no specified final position which is in turn blocked by a movable object that has been moved to its final position. Figure 39 shows a task of this type. The jaws will not be able to get to their final position after they move A.

Tasks for Which This System Will Find Solutions

There are two types of tasks the system can solve. It can be requested to move one object to a specified final position or to move two or more objects to specified final positions.

Task Type: Move One Object to a Specified Final Position

Sufficient Requirements for a Solution:

Sufficient Out of the Way Places

Task in which jaws will not be able to move to their specified final position

Initial Positions

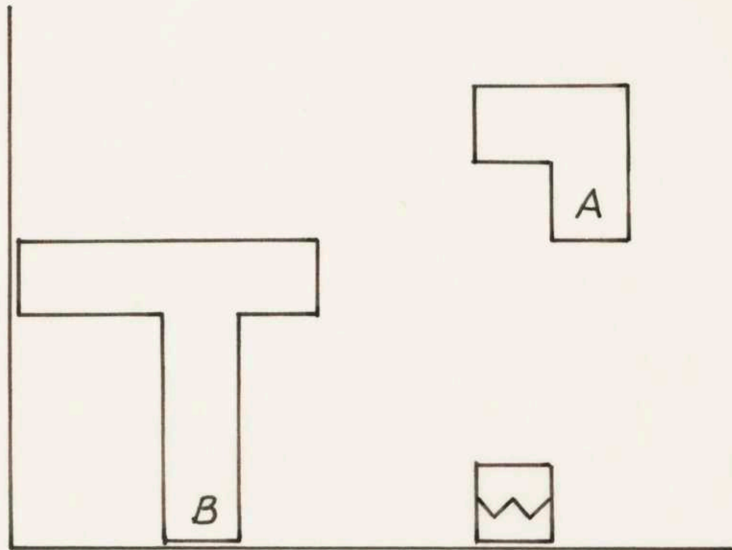
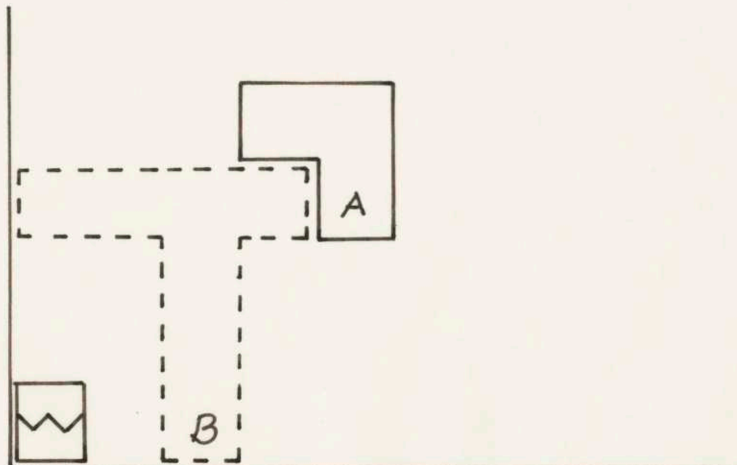


Figure 39-a

Final Positions



The Final Position of B is not specified. It is shown only to emphasize the Final Position of A.

Figure 39-b

How the Solution is Found

The system starts by asking the shortest path algorithm to find a path. The shortest path algorithm returns a value corresponding to a) no solution, b) path found, c) path found but requires moving objects out of the way. For the moment, assume that a) does not occur. If b) is returned, the solution is found. If c), then the shortest path algorithm tries to find paths to Out of the Way Places for each object in the way. For each attempt to find a path, a value corresponding to either b) or c) is returned. Each time c) is returned, at least one sub-task--each requesting an object to be moved out of the way--is put on the TASK TREE. Eventually, for one object the shortest path algorithm must return the value corresponding to b). The system cannot indefinitely return c) as there are a finite number of objects, and the system detects logical loops in the task structure. (The maximum number of times c) can be returned is one less than the number of movable objects in the space.) When the value corresponding to b) is returned, the system can move the object out of the way. The system can then move the previous object (previous object means the object named in the predecessor task on the TREE) out of the way, and continue until it has

moved the object the operator requested. (Remember that moving an object out of the way means it will be moved out of the way of all objects whose paths have been planned, and not just the previous object.)

On the other hand, if a) is returned when planning to move the object requested by the operator, the task is defined is impossible. If a) is returned when planning to move other objects out of the way, the system does not conclude the task is impossible. It attempts instead to find an alternate solution to the problem. The system sets the object it tried to move temporarily immovable. (This object is set movable again when the predecessor chain of the TASK TREE is changed from what it was when the object was set immovable.) This forces the system to find an alternate plan for moving the previous object. This strategy assures that all reasonable combinations of moving objects out of the way will be tried before the system declares the task to be impossible. Therefore, if there is a solution, the system will find it.

Task Type: Move Several Objects to Specified Final Positions

Sufficient Requirements for Solution:

1) Last out - First in rule applies. (It must, as the time sense of the task is reversed.)

2) All Out of the Way Places for objects that have specified final positions are Connected.

3) There are sufficient Out of the Way Places.

<u>How the Solution is Found</u>

Requirement 1 guarantees that the system will find the correct order for moving objects to their final positions.

Requirement 2 guarantees that if an object can be moved from any one particular Out of the Way Place to its final position, then it can be moved to its final position from any Out of the Way Place in the space. Also, as it is assumed that diodes do not exist in the task space, an object can be moved from any Out of the Way Place to its initial position. For a solution to the task to exist, the system must be able to <u>plan</u> a path for the object from its initial position to its final position. Therefore, if an object is in its initial position or any Out of the Way Place, a path can be planned to move the object to its final position.

The above guarantees that paths can be found and will be planned in an order that will solve the task--if

a solution exists. Now, all that remains is to show that each object will be moved to its specified final position. But this is identical to the first task type, and it has already been shown that these solutions will be found. Therefore, the solution to the complex task will be found.

Chapter VI Self-Diagnosis of Failure

Failures occur only in the shortest path algorithm. The shortest path algorithm fails by not finding a path with the required end points. Failures occur, for example, because immovable objects or walls prevent the jaws from grasping or moving an object. Failure can occur either in planning the object and jaw's motion, or in planning only the jaw's motion (jaws moving with object stationary). As presently implemented, the system cannot distinguish where the failure occurs.

The presently implemented version of the system has very limited failure diagnosis features. If a task fails, the system gives the operator a chance to inspect the TASK TREE as it was before the failure. From this inspection, the operator can gain an understanding of the state of the entire task, including the specific Simple Task that caused the failure. If the Simple Task was explicitly requested by the operator, the system decides the entire task has failed, and halts. If the Simple Task was generated by the system, the system attempts to find an alternate solution. It is possible that a solution has been overlooked. To search for this solution, the system sets

the object it tried to move fixed, and continues with the complete task by attempting to execute the next task on the TREE. By doing this, the system will try moving all reasonable combinations of objects before it abandons a task as impossible. The system is designed to attempt these recoveries as it may decide to move the wrong object first.

In addition to this failure procedure, there are others that could be implemented. One possibility, suggested above, is to indicate which part of the shortest path algorithm failed (moving jaws and object, or jaws alone). This would help the operator discover the exact cause of failure.

As another possibility, when the system finds that it must move objects to Out of the Way Places before it finds the paths, it could compute the area required for these Out of the Way Places. If this area were greater than the area available as Out of the Way Places, an appropriate error procedure could be entered.

Also, it might be possible for the system to check that all of the Out of the Way Places are connected for each object that has a specified Final Position.

These failure tests cover all of the possibilities of the system failing, except for the Last out - First in requirement. But the Last out - First in rule must apply as the complete time sense of the task is reversed.

Chapter VII Economic Advantages of the System

A Comparison of a Complete Optimization Method, Whitney's System, and This System

Whitney,[32] in his Chapter 5, discussed the reasons for not using full optimization methods to find paths for planning to move two or more objects. He also outlined a method to find paths for moving more than one object. This method depended on an operator specifying sets of Out of the Way Places for objects. It finds an optimal set of paths (given that the operator specifies the Out of the Way Places) for moving the objects, but at a cost of inconvenience to the operator and of the system having to compute many paths.

Whitney's system and this system will be investigated and compared below to determine which one should be used for greater efficiency in various situations. The time taken to find complete task solutions and the cost of the total paths found will be compared. It is assumed that both systems use the same shortest path algorithms.

The first task investigated is moving one object, i.e., a simple task. Whitney's system will perform better in this task, as it does not have to support the overhead of the second level problem solving system. The paths found will be

the same cost, but the computer time used will be greater for this system. However, the time will be greater by only one-tenth of a second or less (enough time to process 10,000 instructions on a slow computer). Therefore, there is a very slight advantage for Whitney's system in this case.

Second, consider the task of moving one object that requires having to move one other object out of the way. (For example, a doorway blocked by one object.) Whitney's system requires a human operator to specify a set of Out of the Way Places. The number of paths (here one path is the solution to one simple task) Whitney's system computes is $1+2 \cdot n$ where n is the number of locations in the Out of the Way Place set supplied by the operator. The minimal value for n is 1, and the minimal number of paths computed is three. This system computes three paths, at all times, and it could handle all the necessary computations in the second level of the system in less than one second as this case is relatively simple. Presumably, the operator using Whitney's system would require at least one second to specify one Out of the Way Place.

As to the path cost consideration, both methods should find paths that cost the same, as there is no choice of which object to move first. There is the possibility

that the operator using Whitney's system would pick his Out of the Way Place far away from the path, further than the minimum distance necessary, and hence his method would find costlier paths.

If the set of Out of the Way Places contains more than one item, then Whitney's system will be much slower. The ratio of computing time can be approximated as $\frac{1+2\cdot n}{3}$ as the large majority of the time in this system (99% or more) is taken by the path finding algorithm.

For this case then, we find both systems to be equally good, provided the operator of Whitney's system chooses <u>one</u> Out of the Way Place which is the optimal Out of the Way Place. If he picks any other location, or more than one location, the performance of Whitney's system will not be as good as the performance of this system.

Third, consider the task of moving an object that requires having to move two additional objects out of the way. Whitney's system requires computing $1+4n^2$ paths with the minimum value of n equal to 2. This is a minimum of 17 paths to be computed. My system requires computing a maximum of five and a minimum of four paths. The savings in computing time is greater than a factor of 3. This savings

increases rapidly, as the number of Out of the Way Places the operator specifies increases. For n=3, the number of paths found is 37, for n=4, 65, etc.

From a cost of paths standpoint, Whitney's system, if directed by a good operator, may find paths that cost only 4/5 as much as the paths found by the system described in this thesis. This difference, 20%, would not seem to compensate for the difference in computer processing times, 300% plus. This is more than an order of magnitude difference in the percentage differences (20% against 300%). For this situation, it can be concluded that this system is the best to use. Note, that for moving four or more objects, the advantage of using this system increases over that found for moving three objects.

In summary, Whitney's system may have a very small time edge when moving one object. When moving two objects, the systems are about even, given a good operator for Whitney's system. But when moving three or more objects, this system requires only 1/3 or less as much computing time as Whitney's, with the penalty of the paths being at worst 20% more costly than those found by Whitney's system. If choosing a system, one must weigh the advantage of much

reduced computing time against the potential disadvantage of slightly more expensive paths. Additionally, one must remember that the system described in this thesis has the ability to discover whether additional sub-tasks must be solved to solve the requested task.

Chapter VIII Examples Solved by the Demonstration System

This section includes several examples of tasks the demonstration system solved. The system output is an oscilloscope display of the movement of the jaws and objects. The motion is shown as a series of still pictures. The difference between two succeeding frames is usually that the jaws or the jaws and an object have moved one unit. The regular display intervals and the after-images combine to give the appearance of jerky but regular motion. The display frames are numbered sequentially in octal, except that occasionally three numbers are skipped. For example, in Task 1 frame 35 is shown immediately after frame 31. The frame numbers refer to relative computer locations. Locations 32 through 34 contain information necessary to set up the display lists.

The output will be presented here as a sequence of photographs. Not every picture presented on the oscilloscope display will be included. The display frames not shown consist of motion in a straight line or in an obvious path. For example, in Task 1, the motion from frame 35 to frame 47 consists of the jaws moving in the +Y direction, from (5,5) to (5,17). Also, in Task 1 when the jaws move from their position in frame 103 down to D, frame 107, they move around C at (20,21).

Plans that the system makes to discover which objects to move out of the way are indicated in two ways. First, the word "PLAN" appears at the top right of the picture. Second, a ghost image of the jaws and the object being moved shows the plan. Frame 21 of Task 1 has a ghost image of the jaws and object A at (10,20). In a plan no object or the jaws are actually moved. Frame 31 is the last picture in this plan. Frame 35 is the _next_ picture shown by the display. The order for viewing the pictures is to compare the Initial Positions and Final Positions, start with the Initial Positions and go through the numbered frames in order, and finally compare the last numbered frame with the Final Positions.

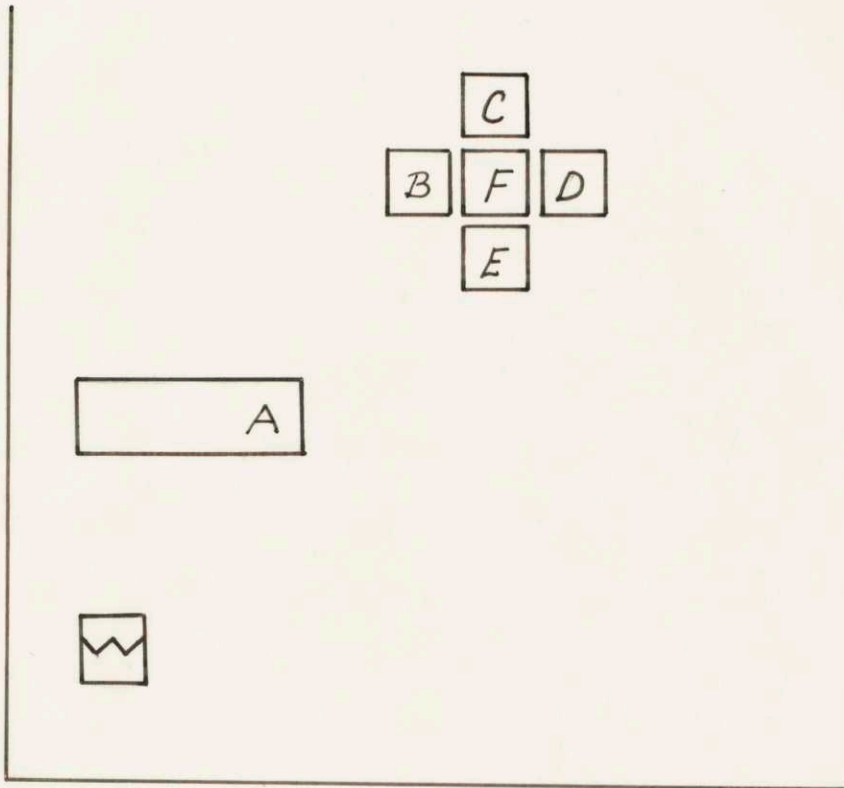
Two Examples Presented in Detail

Task 1

The first task requests moving one object to a specified final position. Figure 40, a representative drawing (not to scale) of the initial positions of the objects, is included as the letters on the objects in the photographs are difficult to read. The final position of object A is the same as the initial position of object F.

Initial Positions

Task 1



Objects A, B, C, D, E, and F.

All objects are movable.

Figure 40

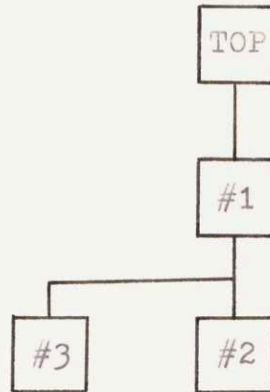
The task proceeds as follows. The jaws move up (frame 5), open and grasp object A (frame 11), move the object A's ghost image up and right to its final position (frames 21, 26, 31).

The system discovers objects B and F are in the way and generates sub-tasks to move them out of the way. The TASK TREE at this point is shown in figure 41. The jaws move up and over and push B up three units, out of the way (frames 35 to 65). The system now plans to move F out of the way, by pushing it across object D (frames 71 to 77). The TASK TREE at this point is shown in figure 42. The jaws move D out of the way of F (frames 103 to 112). F is then moved out of the way (frames 122 to 125). The jaws then grasp A to carry it to its final position (frames 145 to 200). Finally, the jaws move to their specified final position (frames 205 to 211).

There are two aspects of this task that deserve special notice. The first is that in moving A to its final position, the jaws grab it on its left and not on the right as a straight forward minimization would. This happens because the shortest path algorithm starts at the final position (not a Temporary Location--see frame 200), and runs

Task 1

TASK TREE



Interpretive List

task #1 move A to (20,20)

task #2 move B out of the way

task #3 move F out of the way

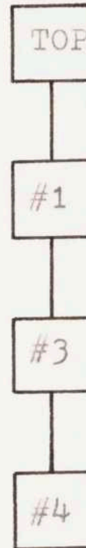
Note: The system executes the bottom right sub-task first; in this case, #2.

The TOP task is a null task, used only as a reference by the system. If the TOP task is the only one left on the TASK TREE, the system knows it has finished the complete task.

Figure 41

Task 1

TASK TREE



Interpretive List

task #1 move A to (20,20)

task #2 move B out of the way

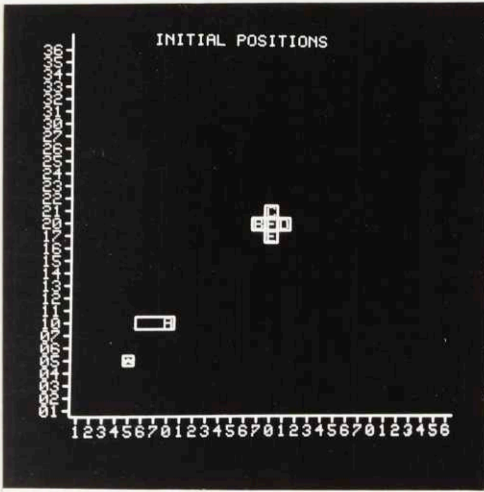
task #3 move F out of the way

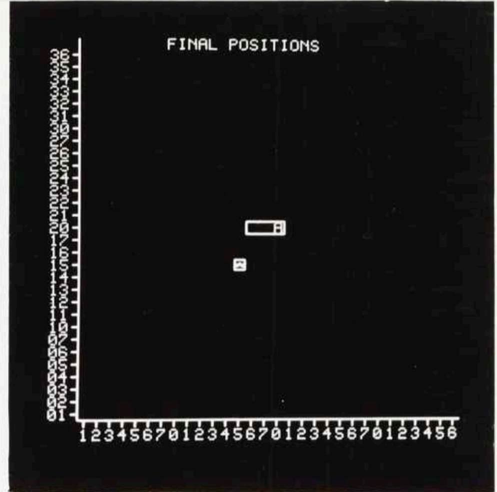
task #4 move D out of the way

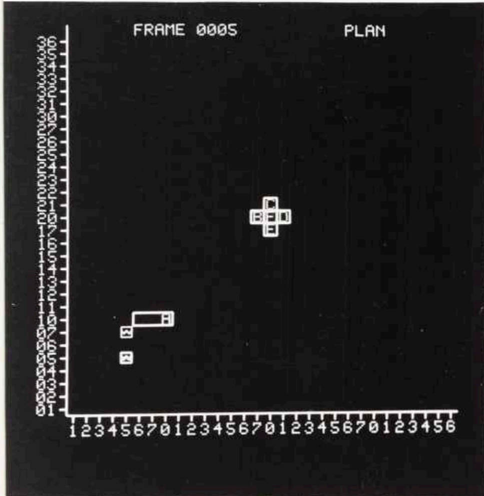
Note: Although task #2 is still on the Interpretive List it will not be executed. Only tasks on the TASK TREE are executed.

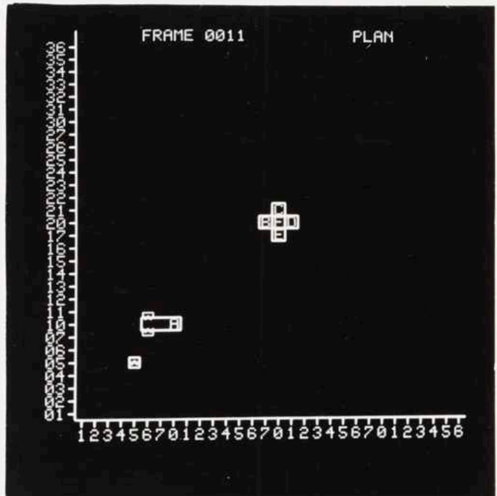
Figure 42

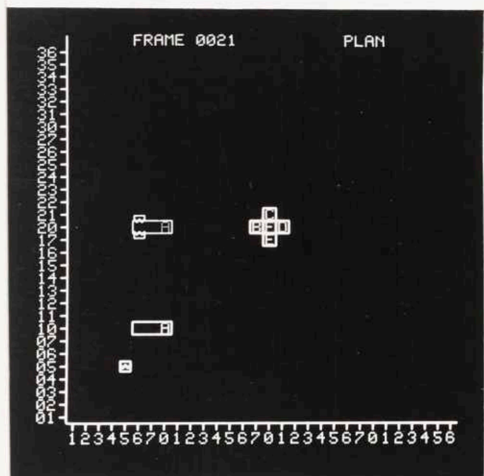
to the initial position (a Temporary Location--see frame 145). The second is the detail of the jaws opening and grasping A, frames 154 to 160. When the jaws grasp an object, the motion is similar to that shown by these four sequential frames.

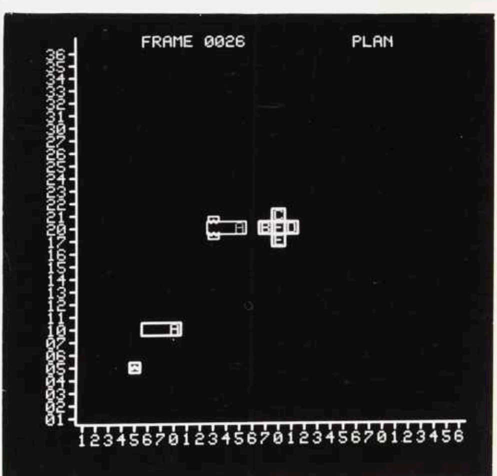

INITIAL POSITIONS

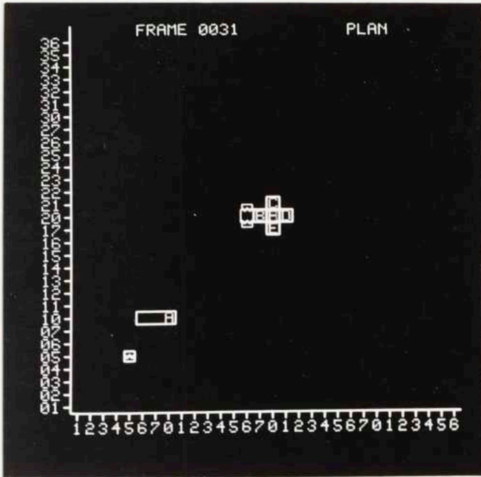

FINAL POSITIONS

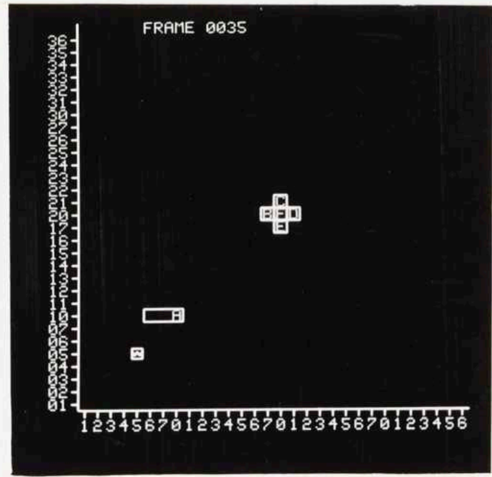

FRAME 0005
PLAN

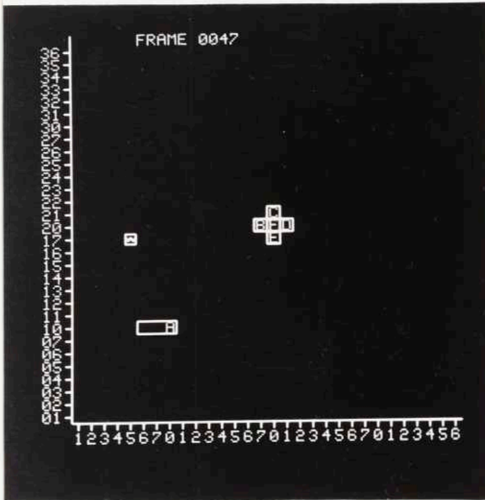

FRAME 0011
PLAN

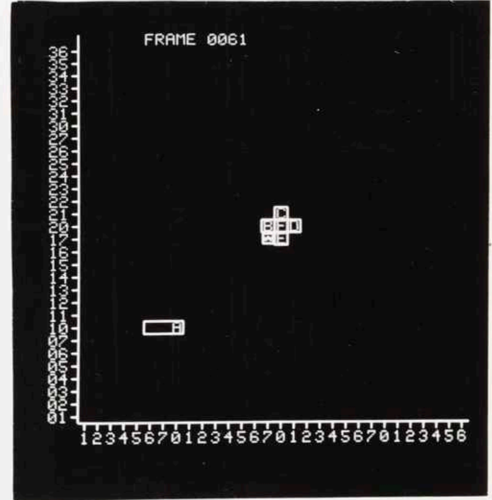

FRAME 0021
PLAN

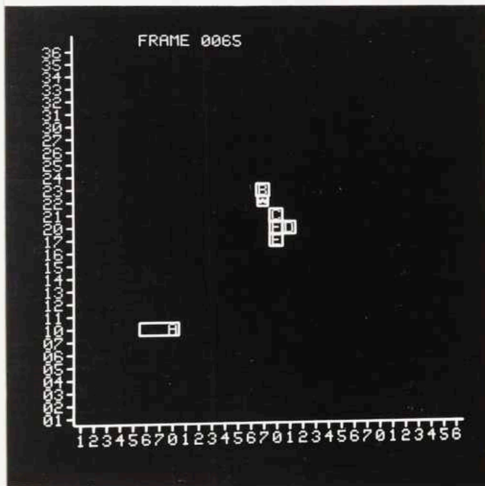

FRAME 0026
PLAN

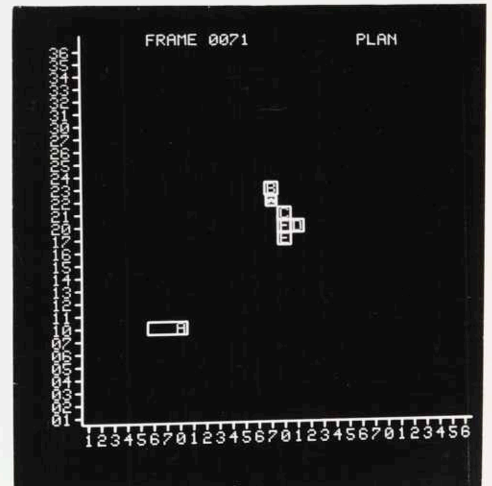

FRAME 0031
PLAN
123456701234567012345670123456

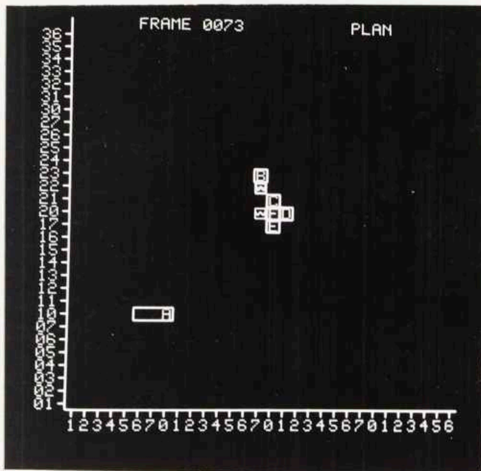

FRAME 0035
123456701234567012345670123456

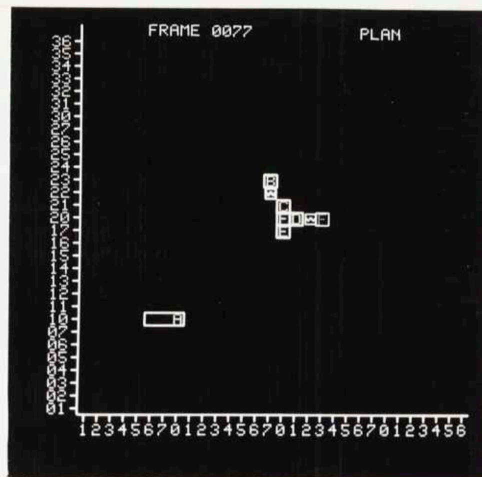

FRAME 0047
123456701234567012345670123456


FRAME 0061
123456701234567012345670123456

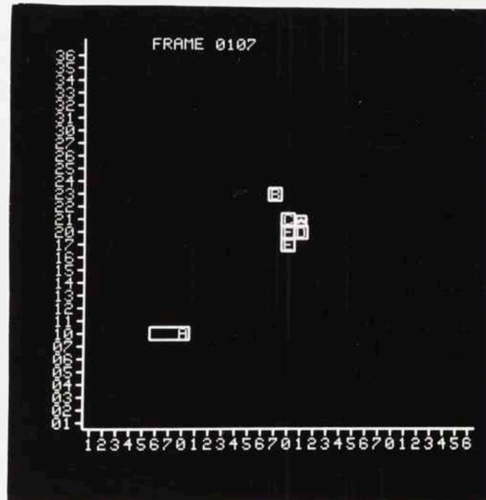

FRAME 0065
123456701234567012345670123456


FRAME 0071
PLAN
123456701234567012345670123456

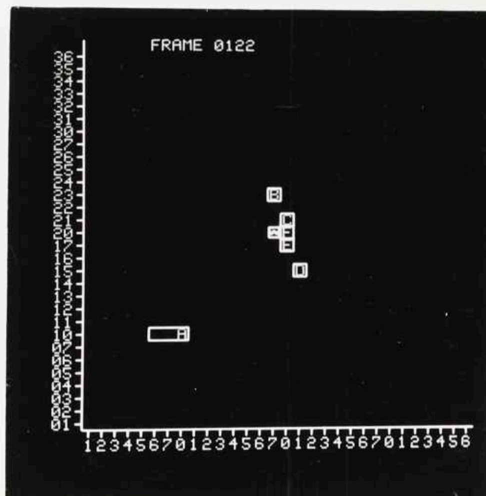

FRAME 0073
PLAN

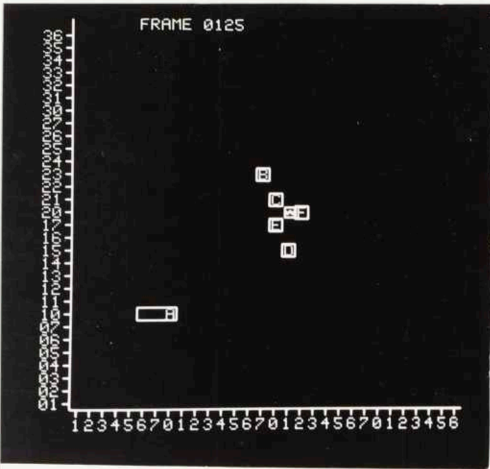

FRAME 0077
PLAN

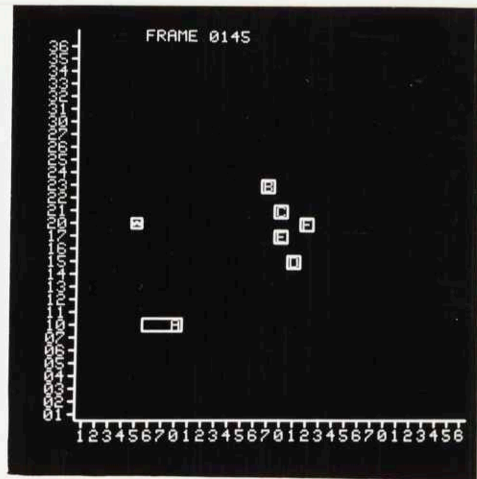
FRAME 0103

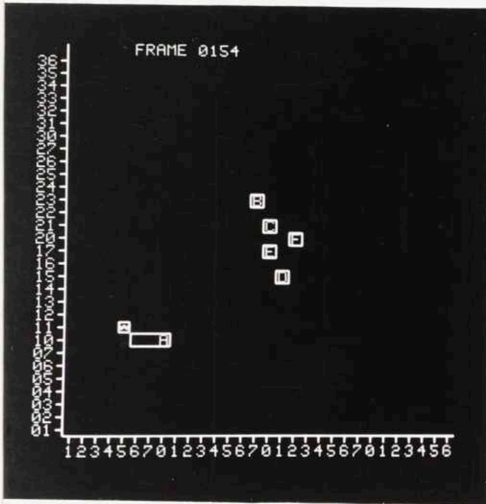

FRAME 0107

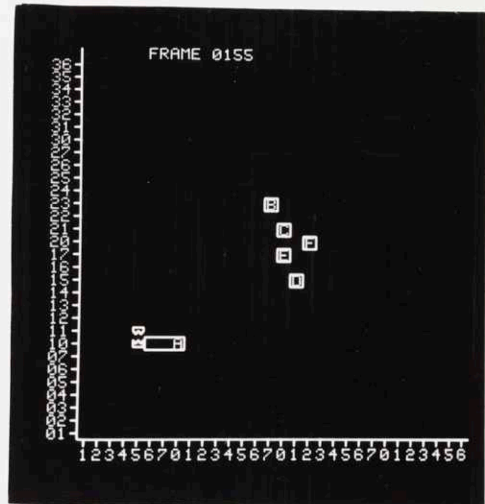
FRAME 0112

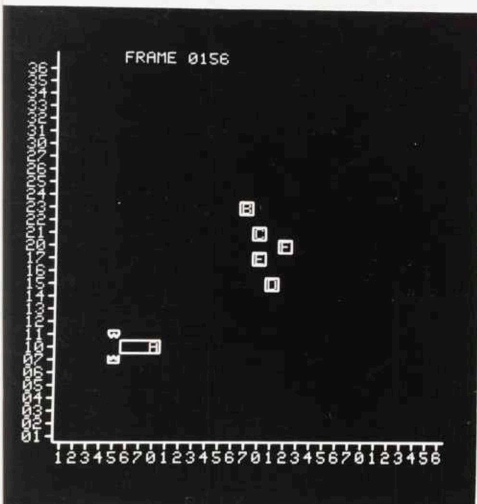

FRAME 0122

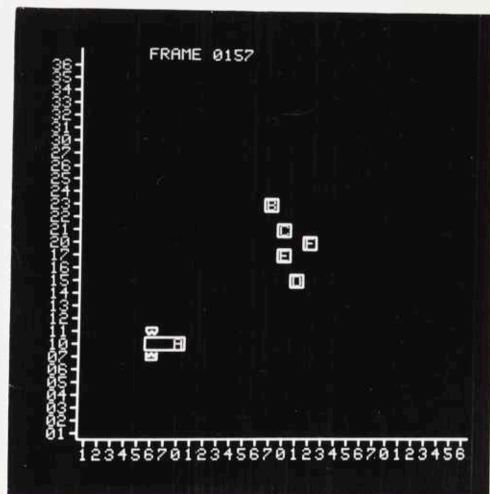

FRAME 0125


FRAME 0145


FRAME 0154


FRAME 0155


FRAME 0156


FRAME 0157

FRAME 0160

FRAME 0170

FRAME 0200

FRAME 0207

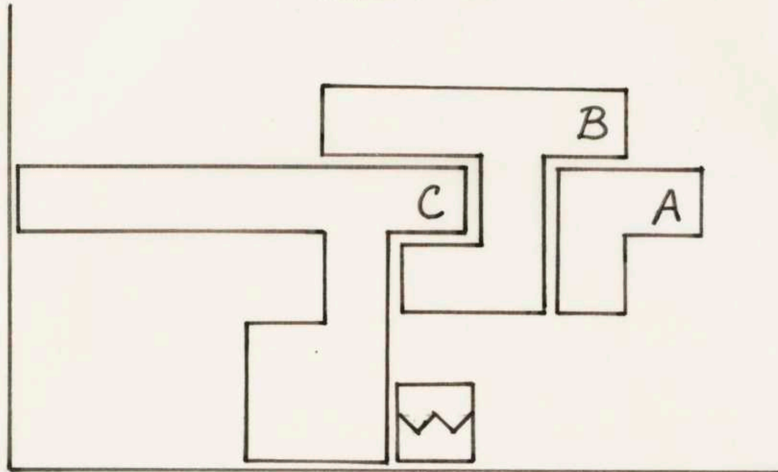
FRAME 0211

Task 2

Task 2 gives the system an opportunity to move several objects to specified final positions. Figure 43-a shows the initial positions and figure 43-b shows the final positions. These are included to aid in identifying the objects in the photographs; neither figure is drawn to scale.

The task begins and the system determines the order to move the objects to their final positions is A first, then B, then C. The system plans to move A to its final position and discovers that C is in the way (frames 11 to 44). The system then plans to move C out of the way and discovers B is in the way (frames 50 to 72). The system then plans to move B out of the way and discovers A is in the way (frames 76 to 126). The TASK TREE at this point is shown in figure 44. The system now moves A out of the way (frames 132 to 151). Note that this is the first time an object has actually been moved in this task. The system then moves B out of the way (frames 160 to 176), followed by C (frames 205 to 234). At this point, only sub-tasks #1, #2, and #3 remain on the TASK TREE, the others having been successfully executed and removed.

The system continues and moves A to its final position (frames 243 to 272), followed by B (frames 301 to 341;



All objects are movable

Figure 43-a

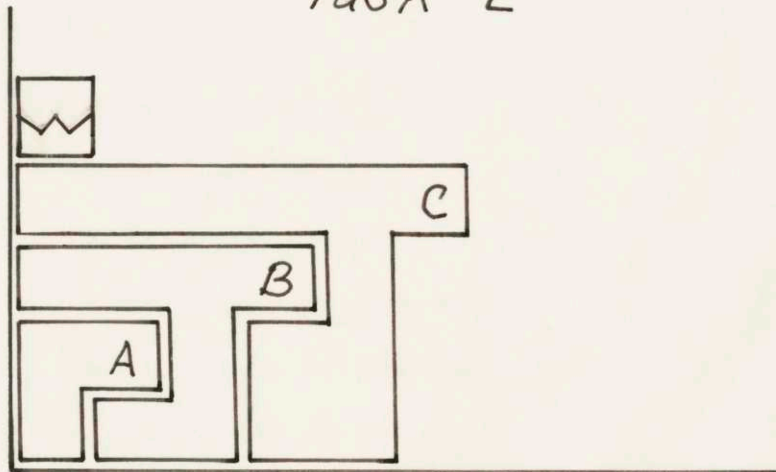


Figure 43-b

Task 2

TASK TREE

Interpretive List

task #1   move C to (6,4)

task #2   move B to (4,3)

task #3   move A to (2,2)

task #4   move C out of the way

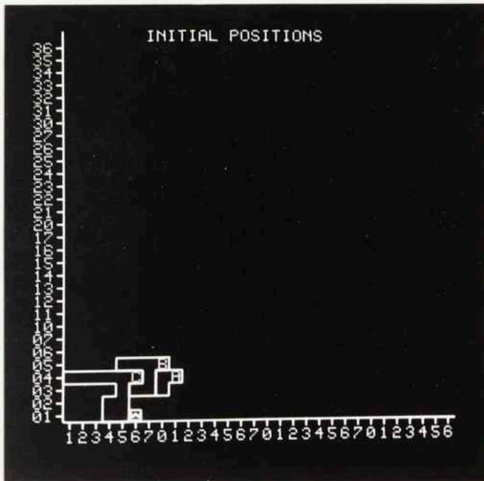
task #5   move B out of the way

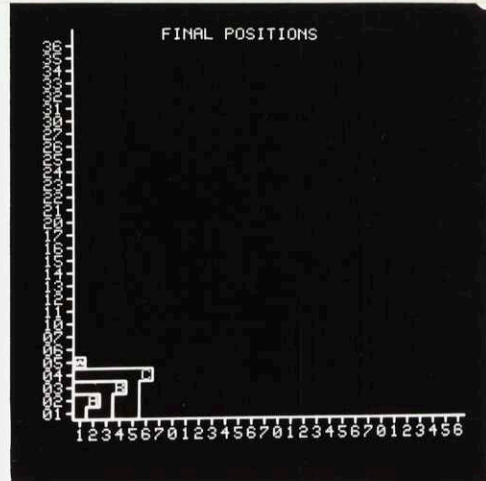
task #6   move A out of the way

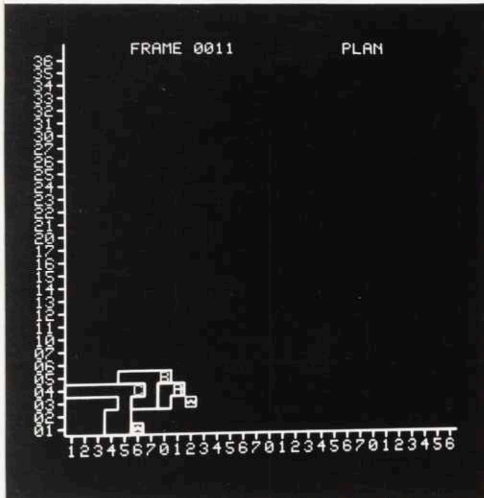
Note: The sub-task, "move q out of the way" is not the same as the sub-task "move q to position (x,y)."

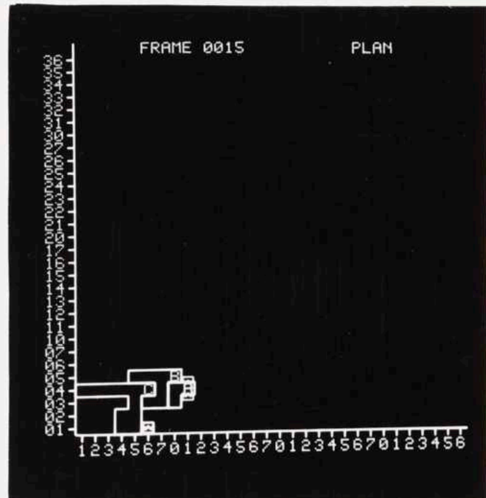
Figure 44

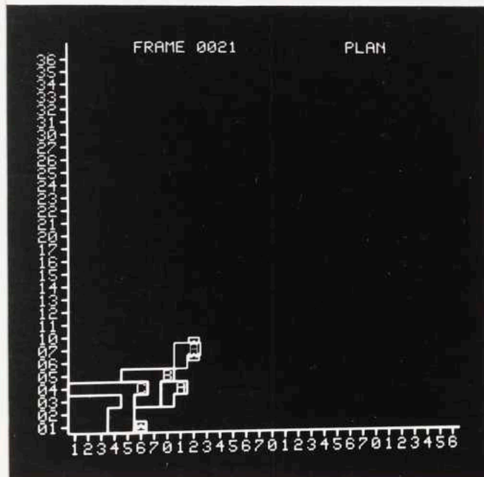
note in frame 301 jaws close at (3,2) for minimum cost path to (20,4)), and then C (frames 350 to 377). The jaws release C and move to their specified final position at (1,5) (frames 406 to 414).

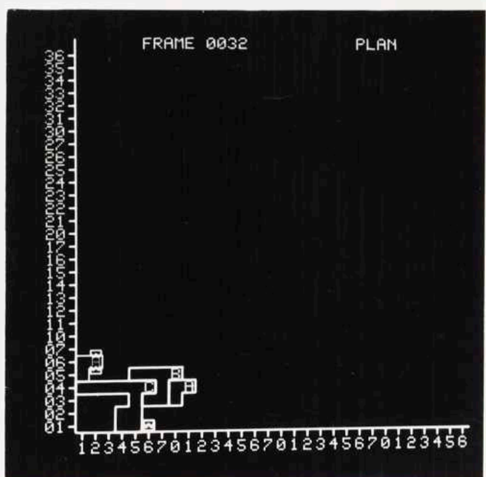

INITIAL POSITIONS


FINAL POSITIONS


FRAME 0011
PLAN


FRAME 0015
PLAN


FRAME 0021
PLAN


FRAME 0032
PLAN

FRAME 0040 PLAN

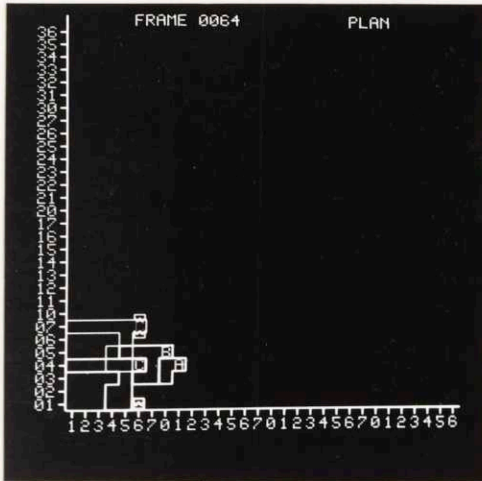
FRAME 0041 PLAN

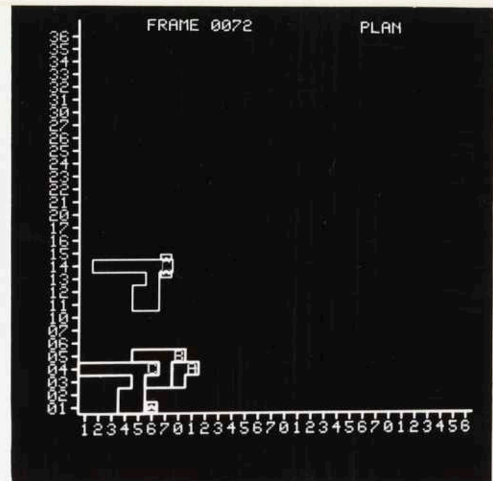
FRAME 0044 PLAN

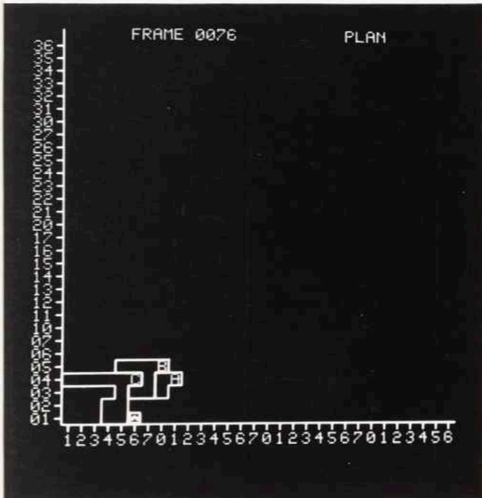
FRAME 0050 PLAN

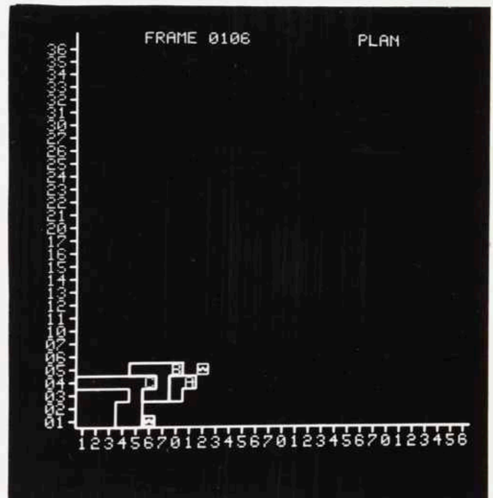
FRAME 0053 PLAN

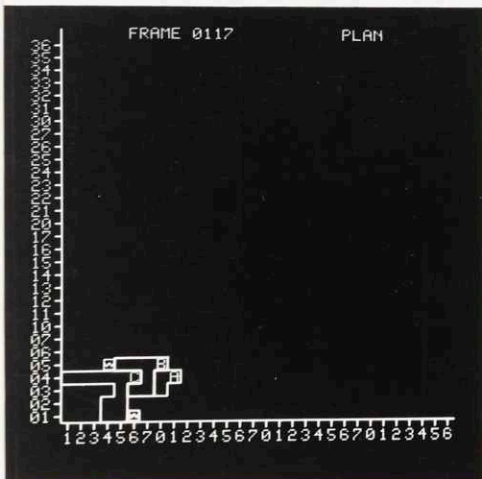
FRAME 0056 PLAN

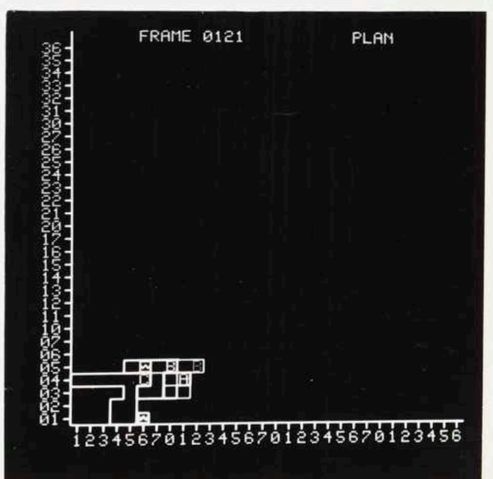

FRAME 0064
PLAN


FRAME 0072
PLAN

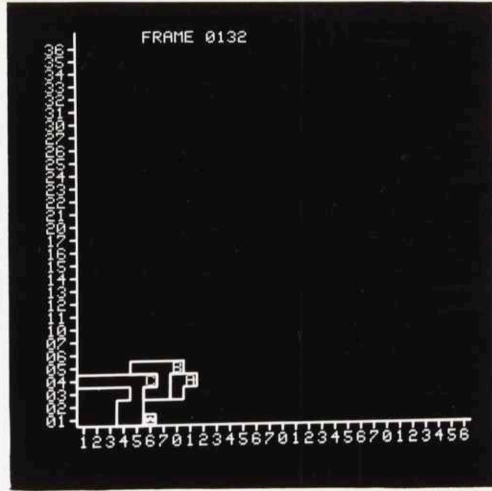

FRAME 0076
PLAN


FRAME 0106
PLAN

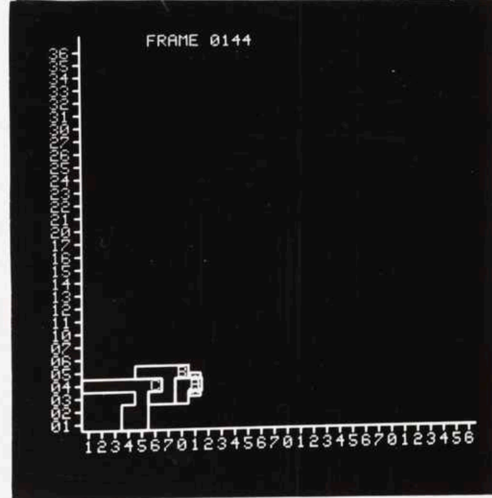

FRAME 0117
PLAN

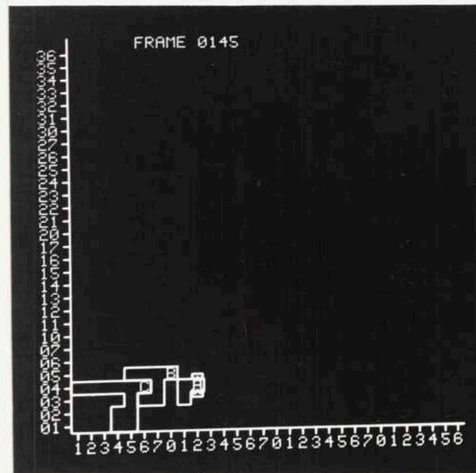

FRAME 0121
PLAN

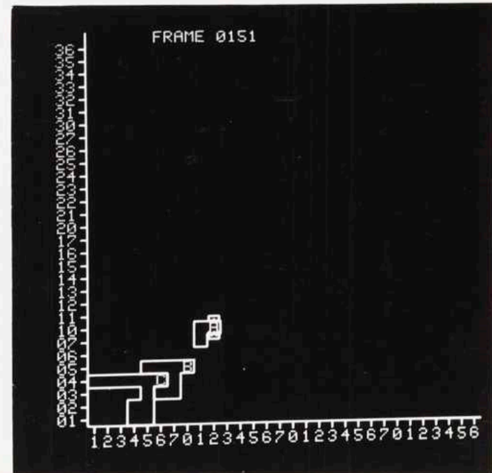
FRAME 0126
PLAN

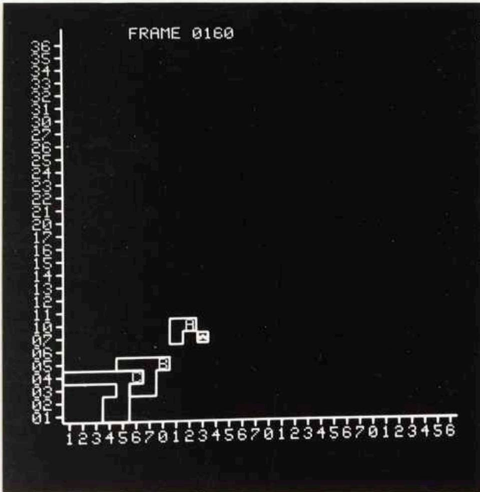

FRAME 0132

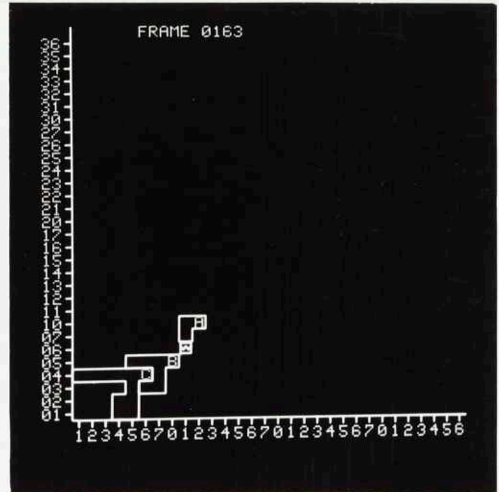
FRAME 0140

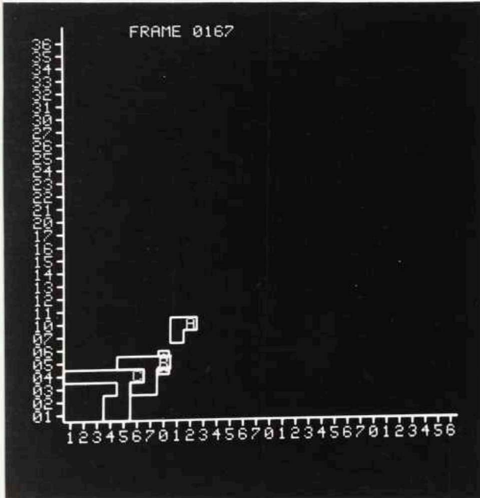

FRAME 0144

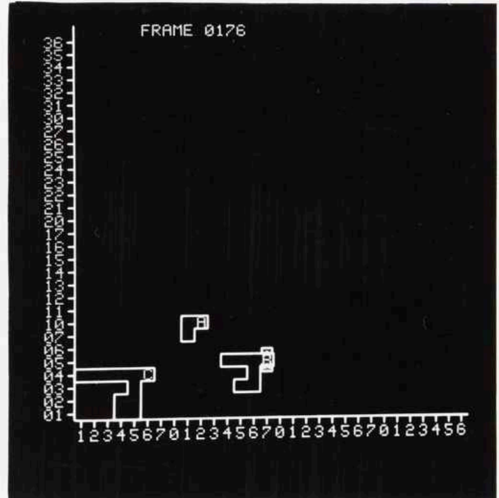

FRAME 0145

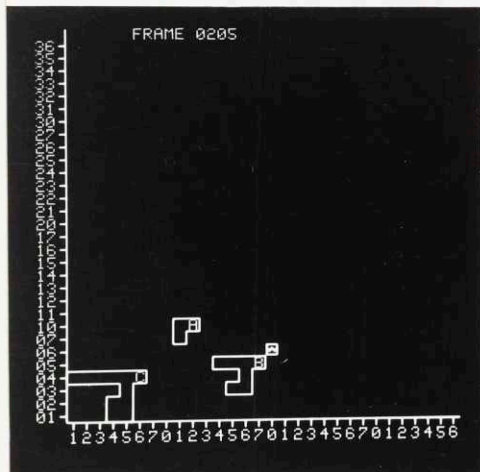

FRAME 0151

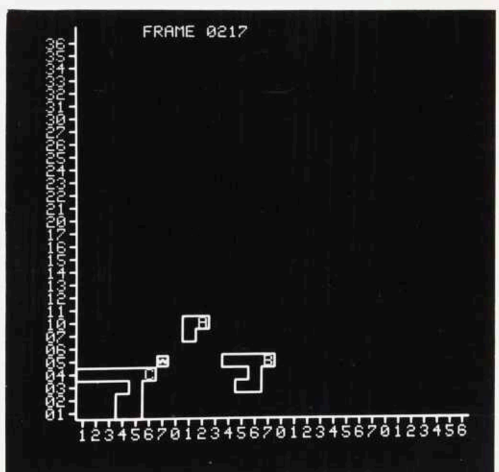

FRAME 0160

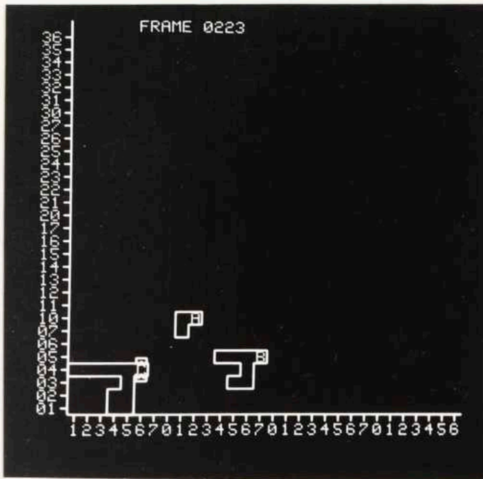

FRAME 0163

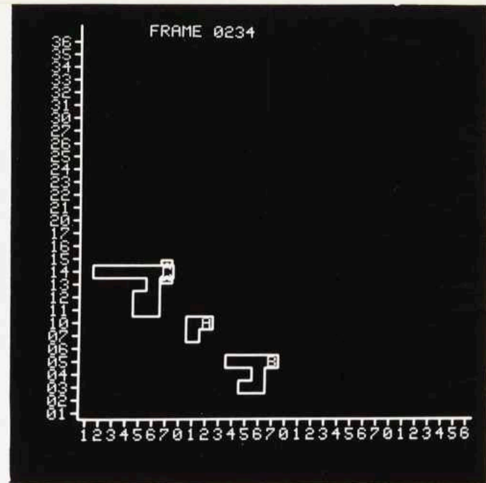

FRAME 0167


FRAME 0176

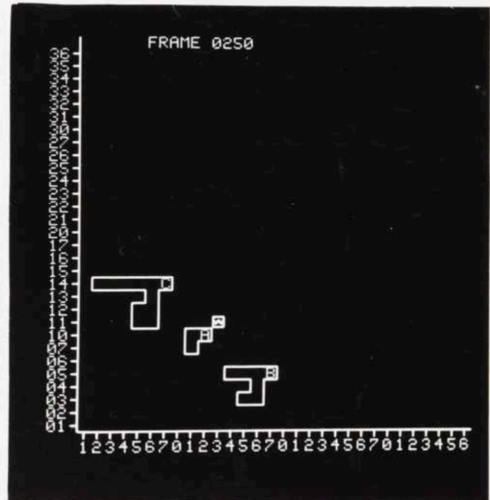

FRAME 0205

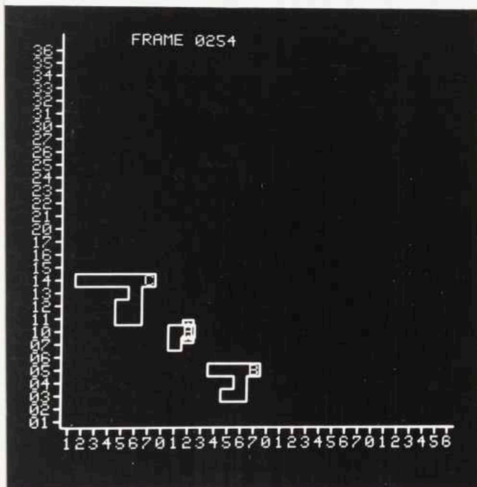

FRAME 0217

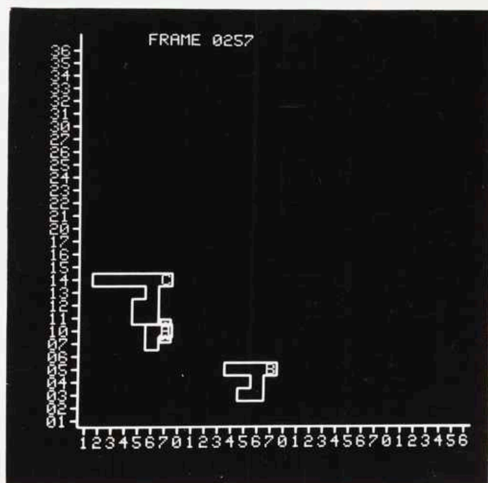

FRAME 0223

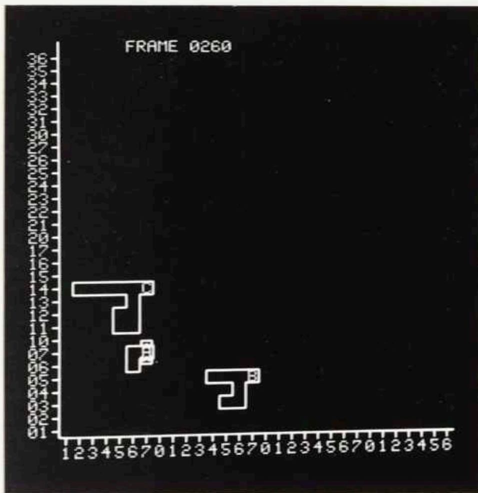

FRAME 0234

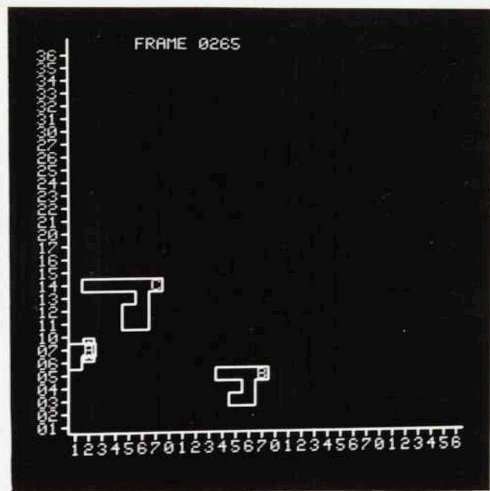
FRAME 0243

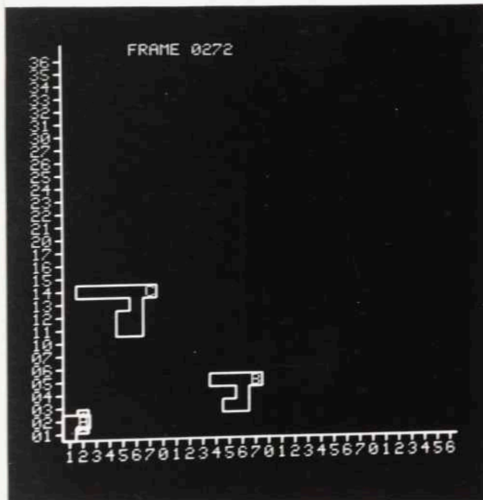

FRAME 0250

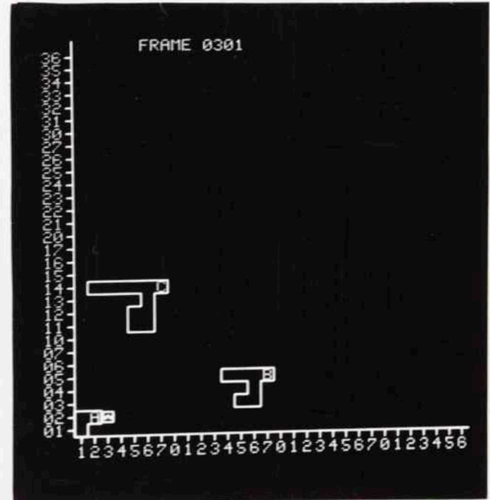

FRAME 0254

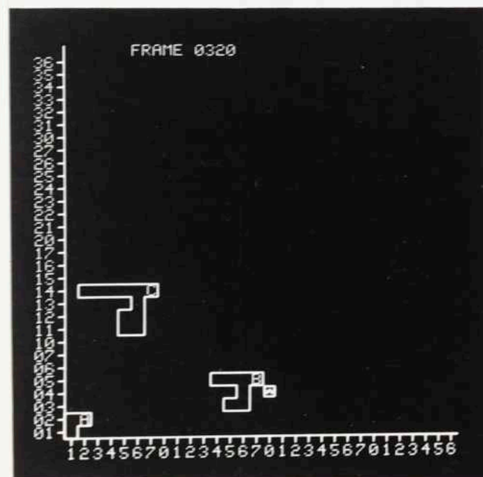

FRAME 0257

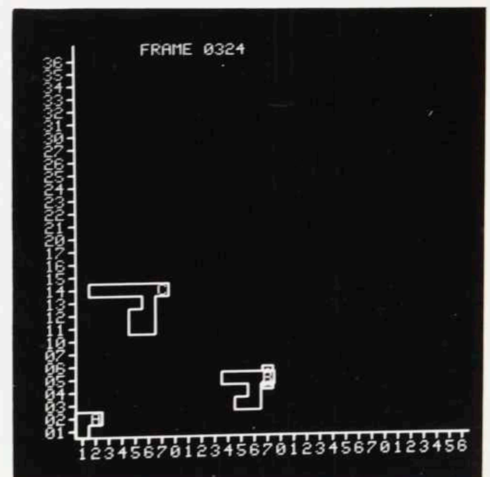

FRAME 0260

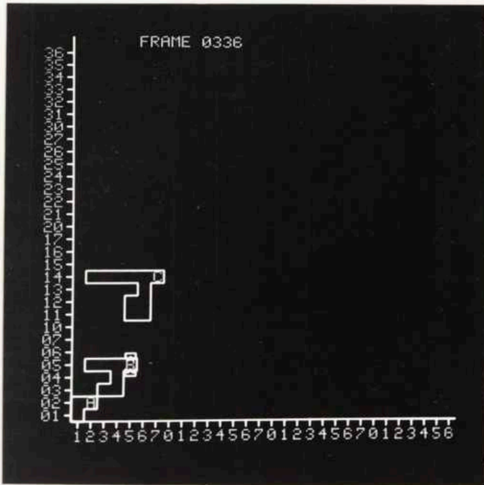

FRAME 0265

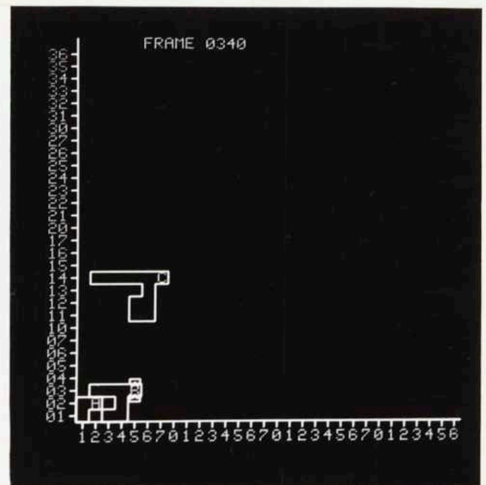

FRAME 0272

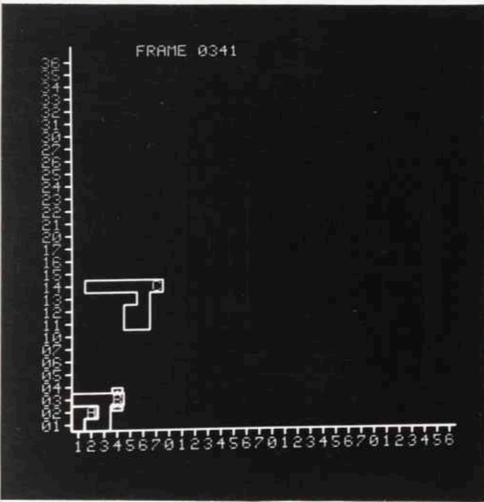

FRAME 0301

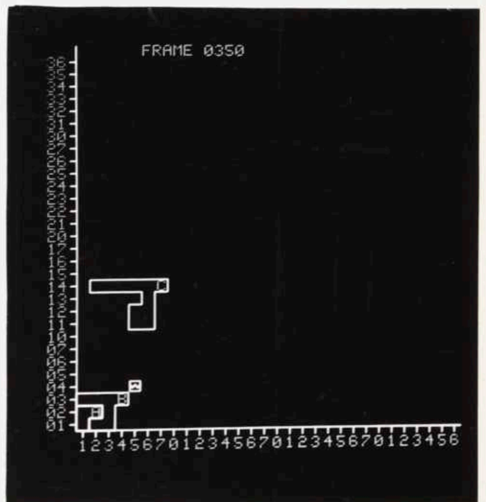

FRAME 0320

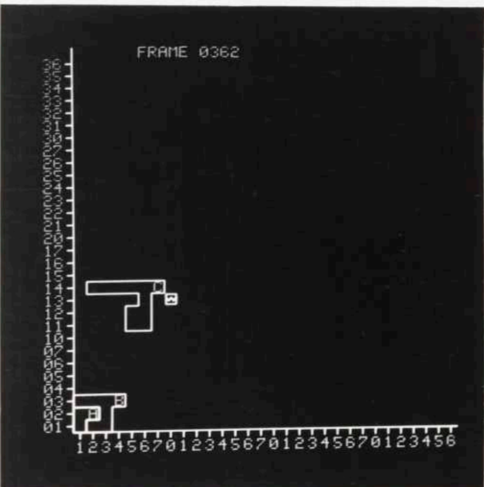

FRAME 0324

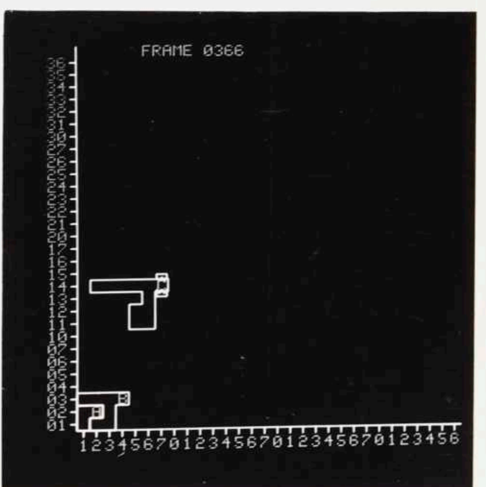

FRAME 0336

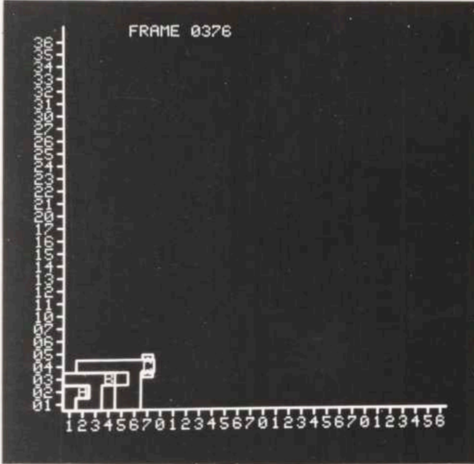

FRAME 0340

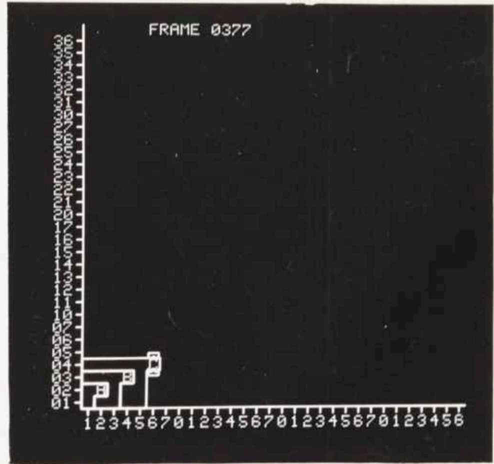

FRAME 0341

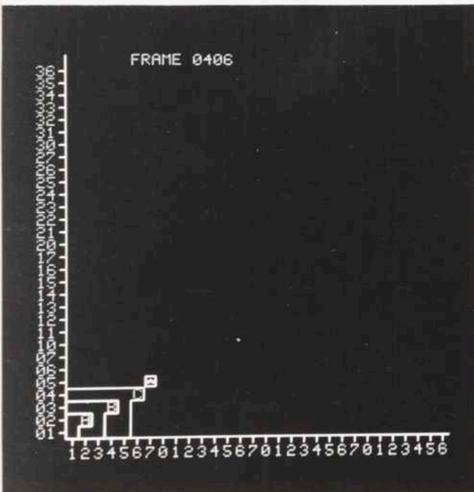

FRAME 0350

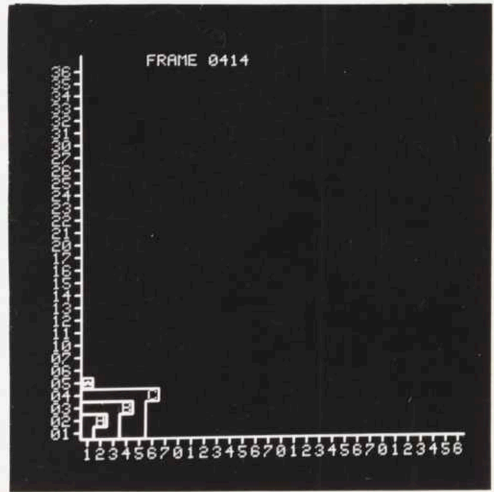

FRAME 0362


FRAME 0366


FRAME 0376


FRAME 0377


FRAME 0406


FRAME 0414

Four Examples Briefly Presented

The following tasks are shown in much less detail than the previous two. The pictures of the first two tasks should have given the reader a good feeling for the paths. These latter tasks are included to give the reader some idea of the types of tasks the system has solved.

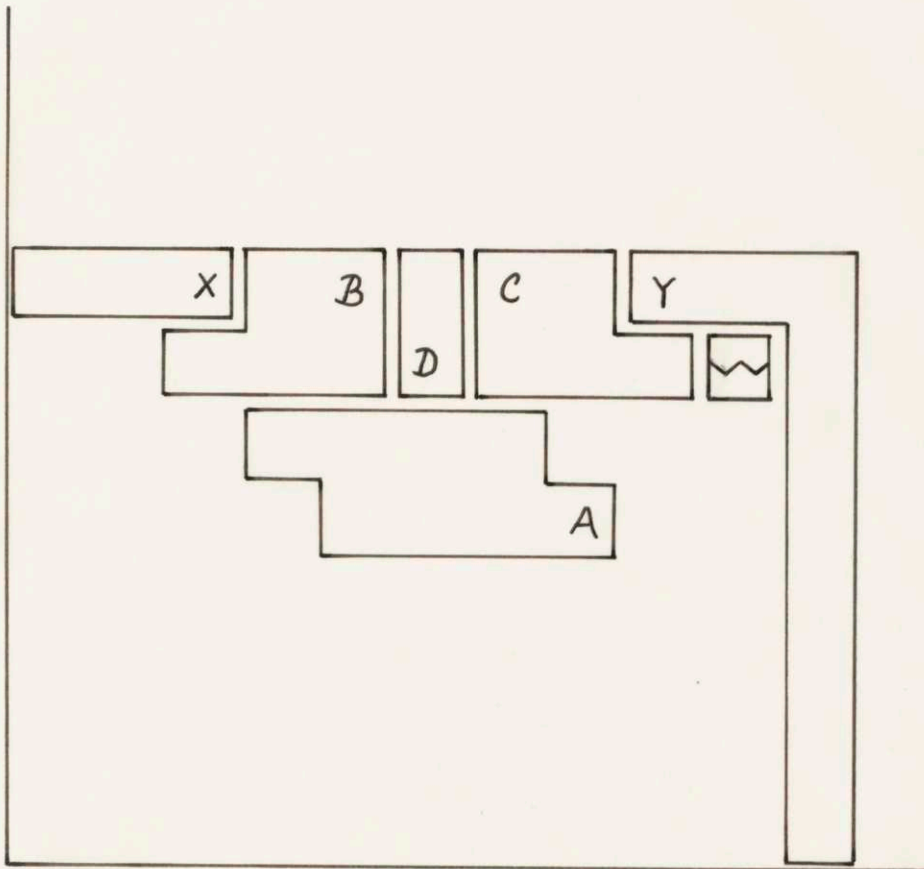
Task 3

Task 3 is called a "locked doorway" after Whitney's[31] "blocked doorway." The initial positions are shown in figure 45. Objects X and Y are immovable.

The task proceeds as follows. A plan is made to move A up and then to its final position during which the system discovers objects C, D, and B are in the way (frames 14 to 26). The system plans to move C out of the way and finds D in the way (frames 41 and 46). The system then plans to move D out of the way and discovers A is in the way (frames 60 and 74). The TASK TREE at this stage is shown in figure 46.

The system moves A out of the way (frame 105) and then D (frame 131). At this point the TASK TREE is as shown in figure 47. The system then moves C out of the way

Initial Positions
Task 3



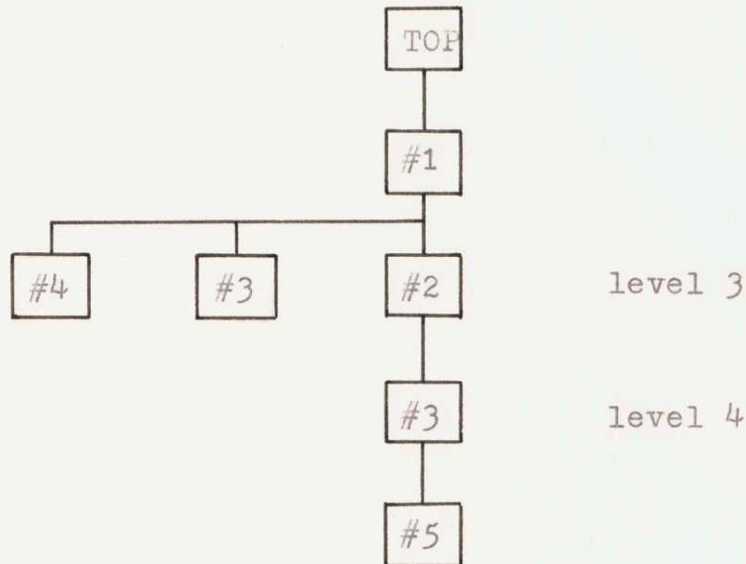
Objects A, B, C, and D are movable.

Objects X and Y are immovable.

Figure 45

Task 3

TASK TREE



Interpretive List

task #1 move A to (12,12)

task #2 move C out of the way

task #3 move D out of the way

task #4 move B out of the way

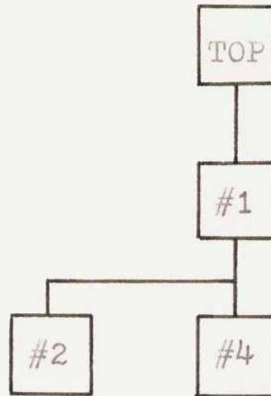
task #5 move A out of the way

Note: Sub-task #3 appears on the TASK TREE twice. The first time, at level 3, it requests D to be moved out of the way of A's path. The second time, level 4, it requests D to be moved out of the way of A's planned path and C's planned path. After sub-task #3 at level 4 is executed, both it and the occurrence of sub-task #3 at level 3 will be removed. See figure 47.

Figure 46

Task 3

TASK TREE

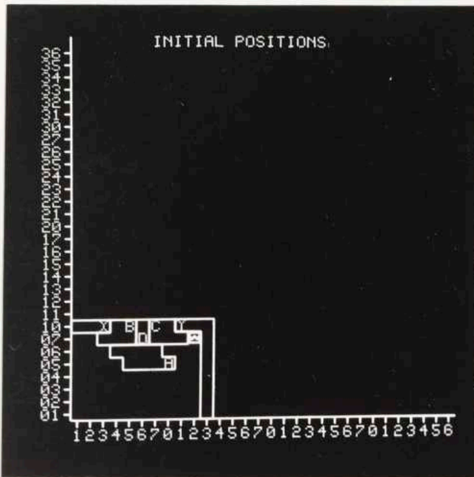


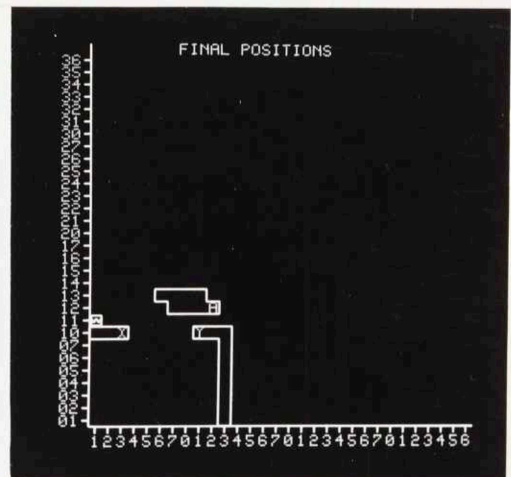
Interpretive List

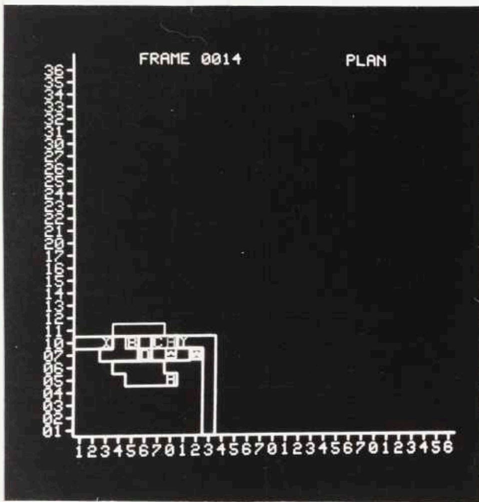
task #1	move A to (12, 12)
task #2	move C out of the way
task #3	move D out of the way
task #4	move B out of the way
task #5	move A out of the way

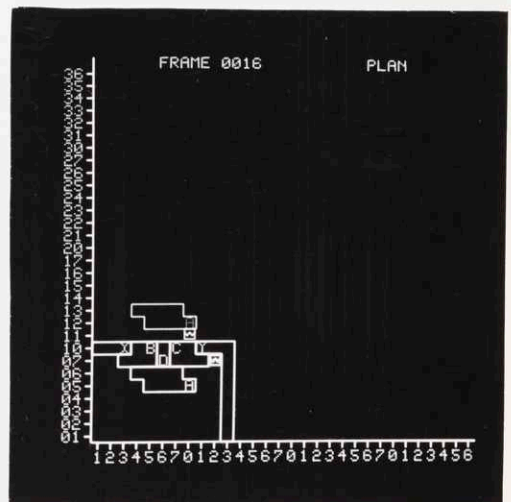
Figure 47

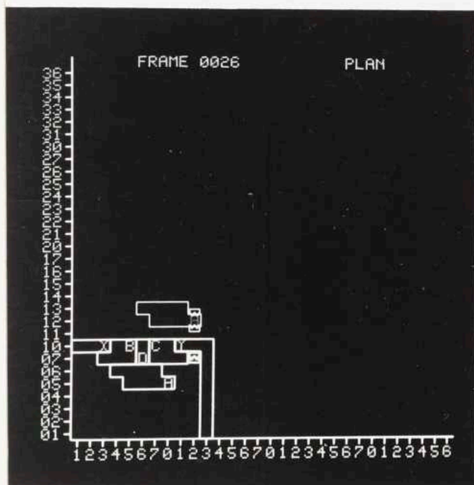
(frames 160 to 166) followed by B (frames 212 to 231). The system then moves A to its specified final position (frame 267) and the jaws move to their final position (frame 303).

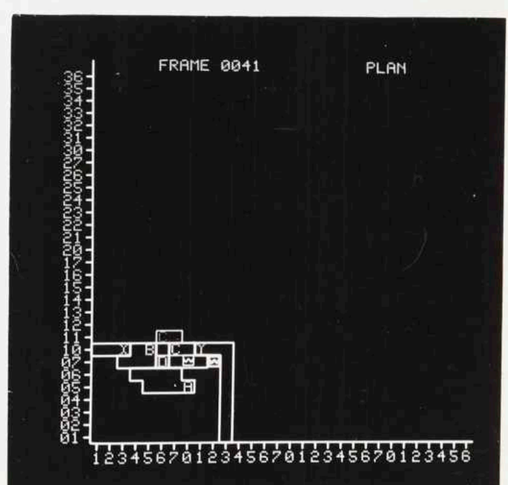

INITIAL POSITIONS

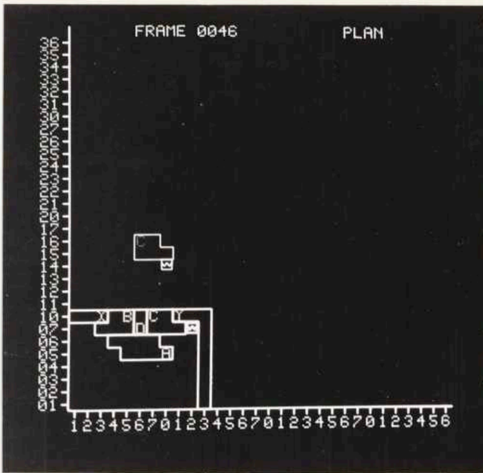

FINAL POSITIONS

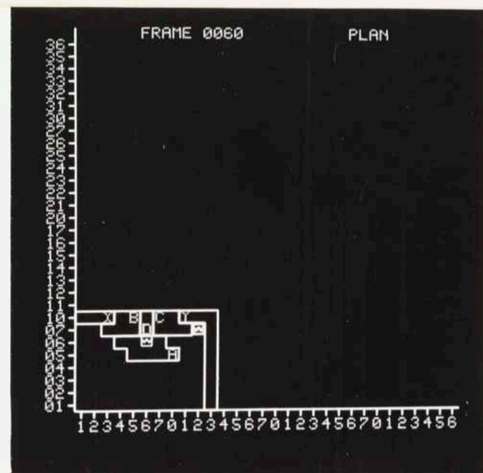

FRAME 0014
PLAN

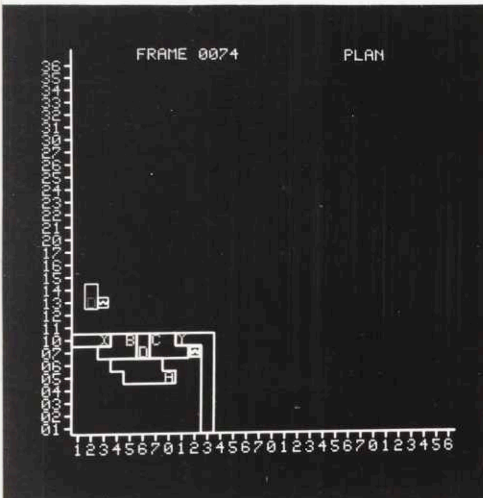

FRAME 0016
PLAN

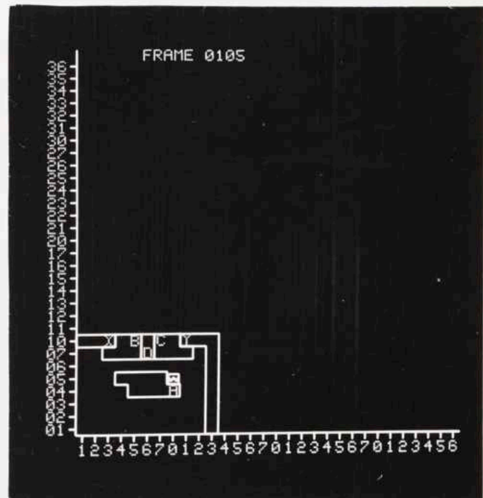

FRAME 0026
PLAN

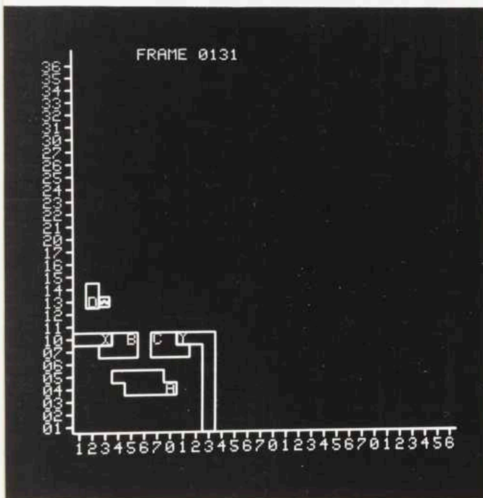

FRAME 0041
PLAN

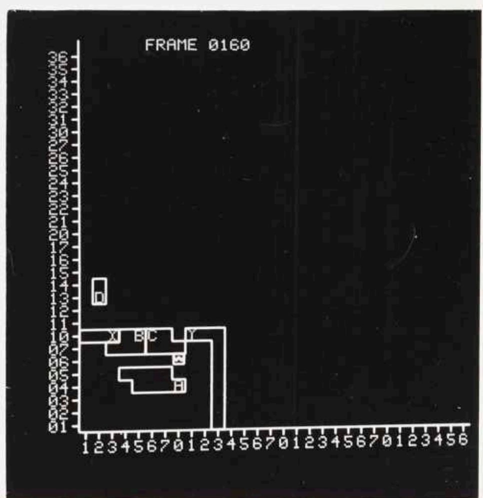

FRAME 0046
PLAN

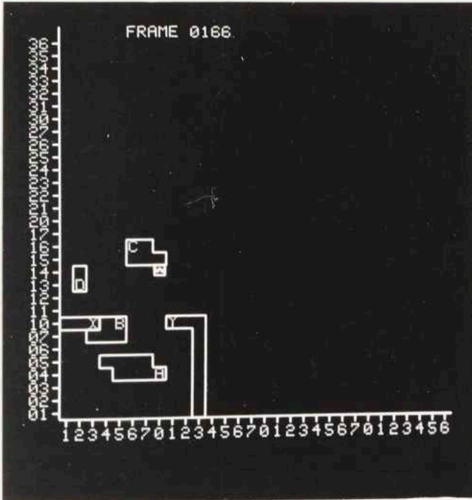

FRAME 0060
PLAN

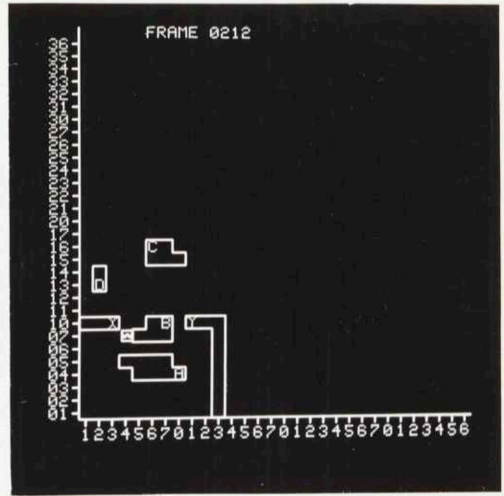

FRAME 0074
PLAN

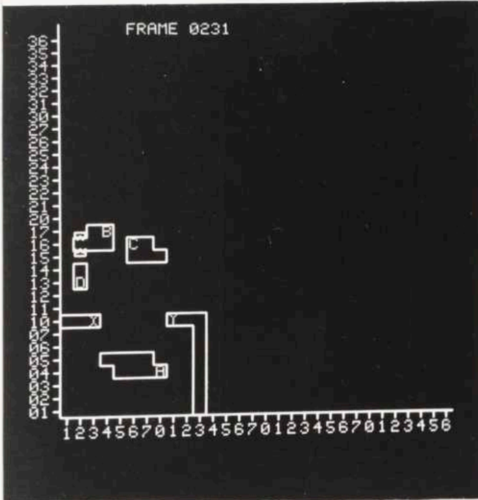

FRAME 0105

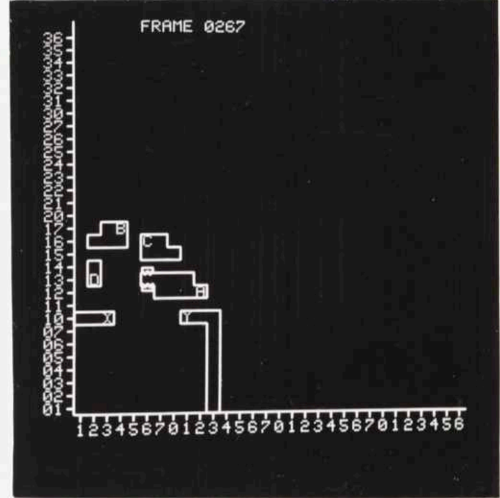

FRAME 0131

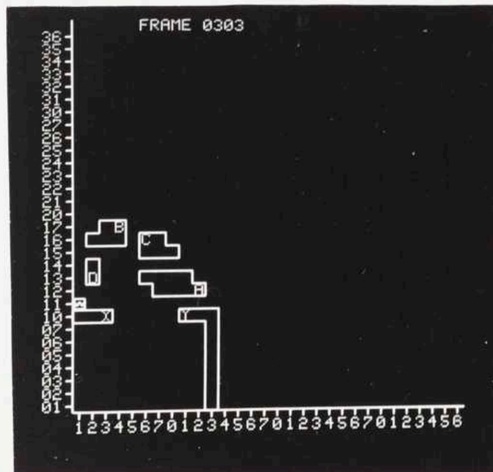

FRAME 0160


FRAME 0166


FRAME 0212

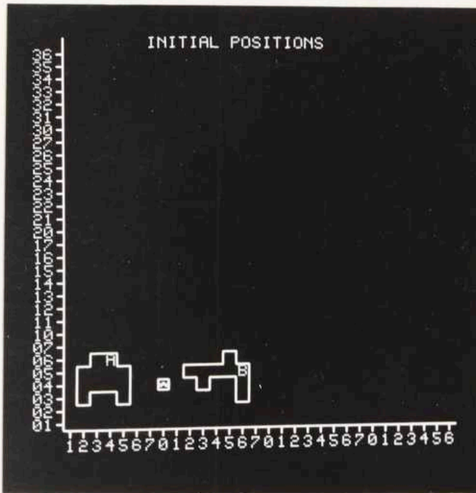

FRAME 0231

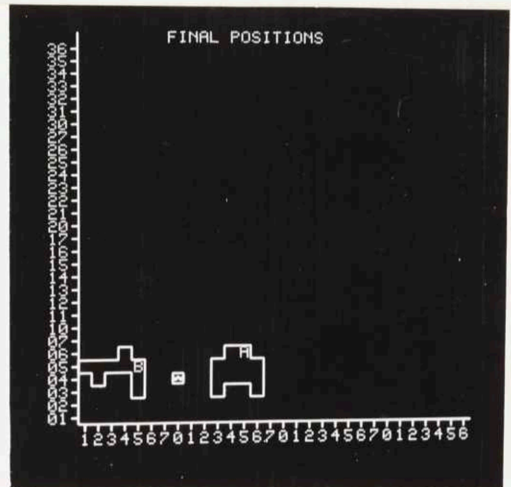

FRAME 0267

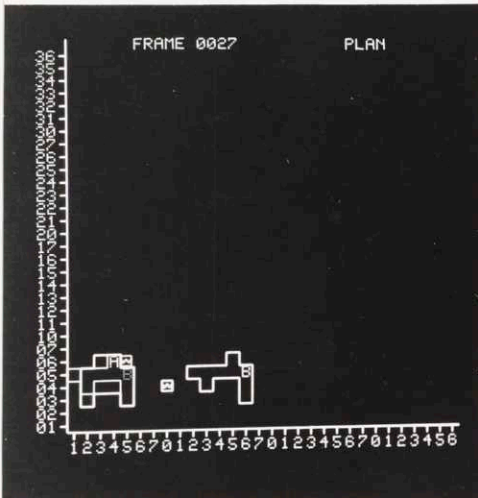

FRAME 0303

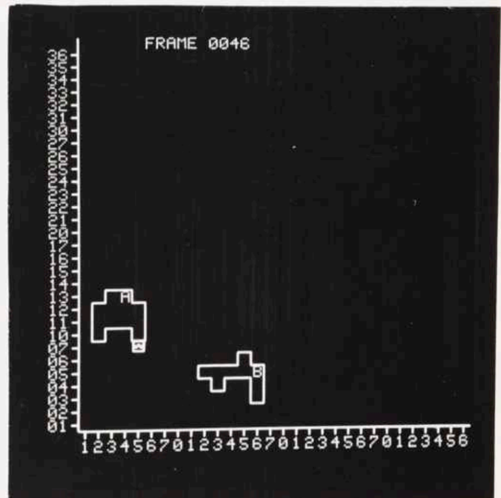
Task 4

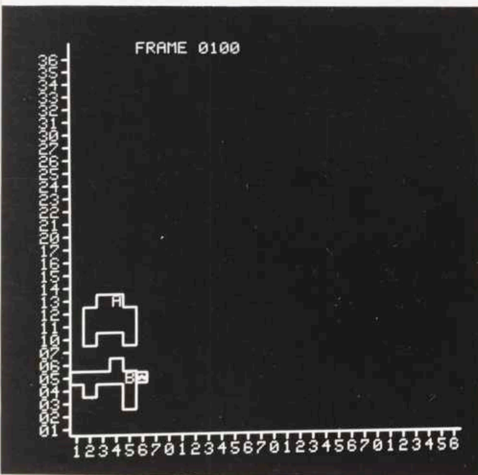
The object of Task 4 is to switch the positions of objects A and B. The system decides the order to move the objects to their final positions is B, then A. The system plans to move B to its final position and finds A in the way, (frame 27). It then moves A out of the way (frame 46), moves B to its final position (frame 100), moves A to its final position (frame 145), and directs the jaws to their final position (frame not shown as it is identical to the Final Position picture).

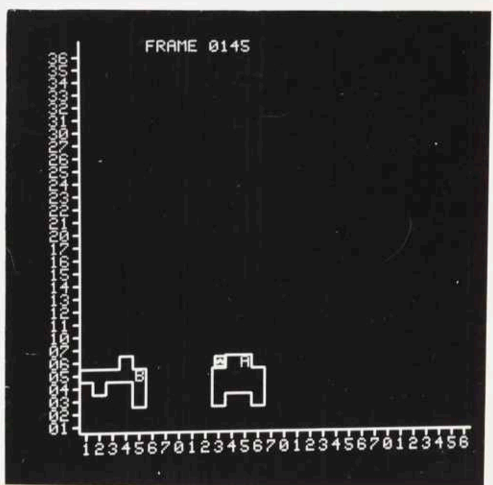

INITIAL POSITIONS
A
B
1234567012345670123456701234567


FINAL POSITIONS
A
B


FRAME 0027
PLAN
A
B


FRAME 0046
A
B

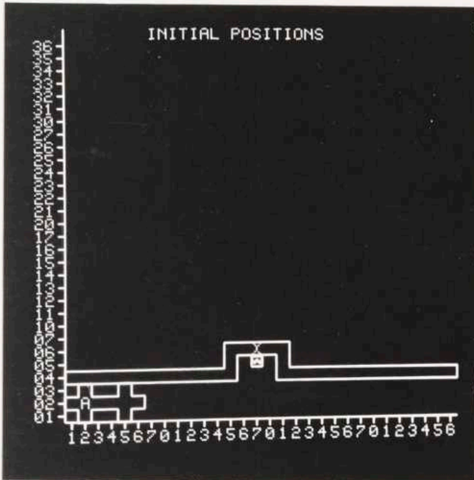

FRAME 0100
A
B

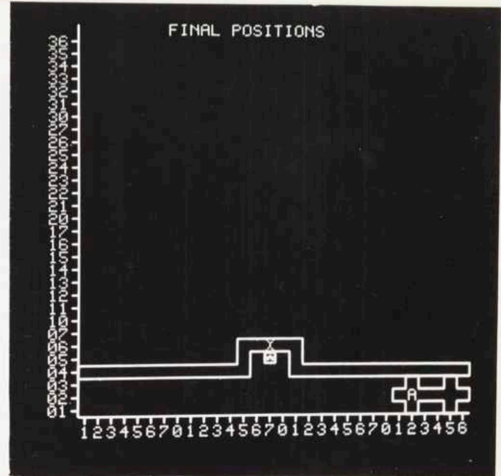

FRAME 0145
A
B

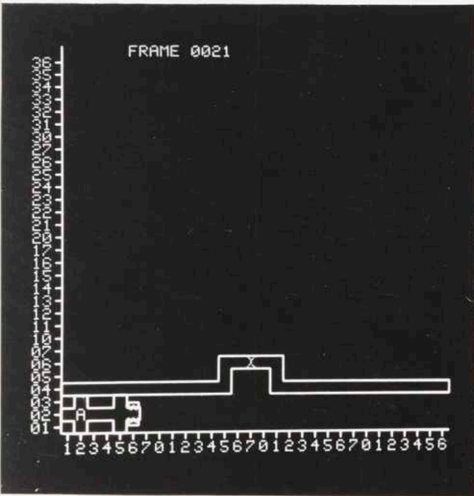
Task 5

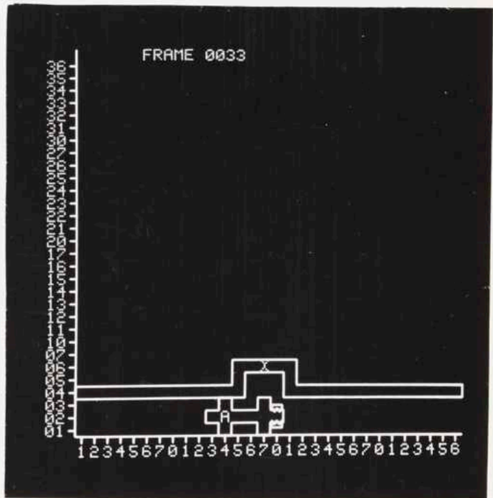
Task 5 is presented to show some of the abilities of the implemented version of the shortest path algorithms, the lower level of the system. Object A is movable, X is fixed. This task could have been solved without the upper level of the system.

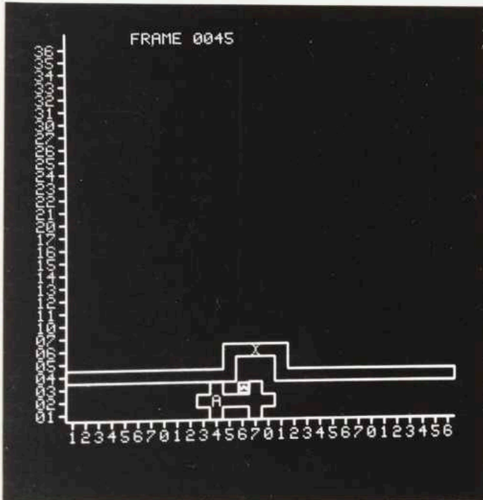
The task is accomplished as follows. The jaws grasp A's right protrusion (frame 21), move A to (14,2) (frame 33), move around to push (frame 45), push A to (17,2) (frame 50), move around and grasp the left protrusion (frame 63), move A to its specified final position (frame 76), and finally the jaws move to their final position. The last frame is not shown as it is identical to the Final Positions picture.

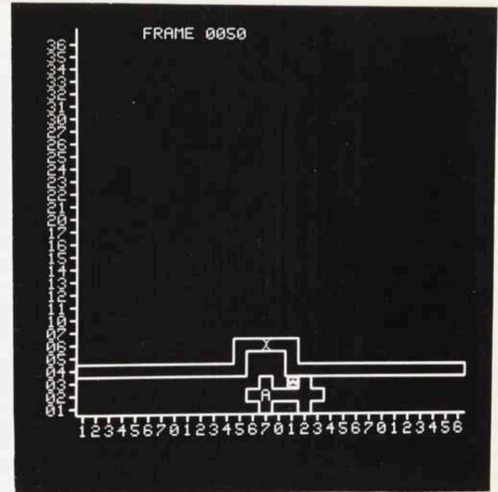

INITIAL POSITIONS
1234567012345670123456701234560

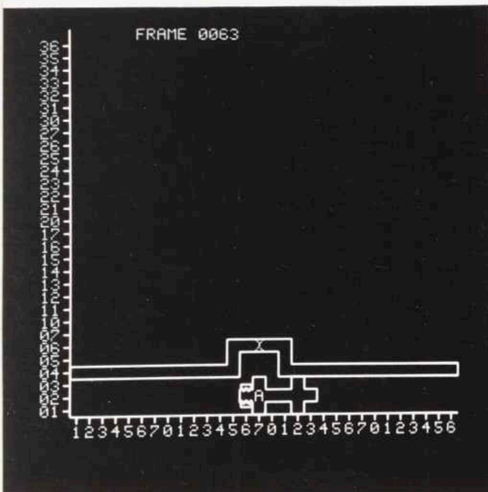

FINAL POSITIONS
1234567012345670123456701234560

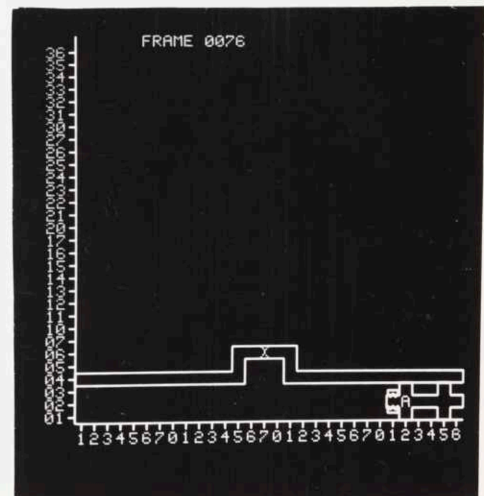

FRAME 0021
1234567012345670123456701234560


FRAME 0033
1234567012345670123456701234560


FRAME 0045

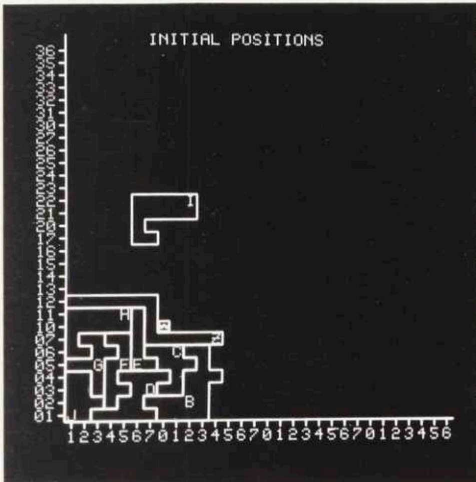

FRAME 0050

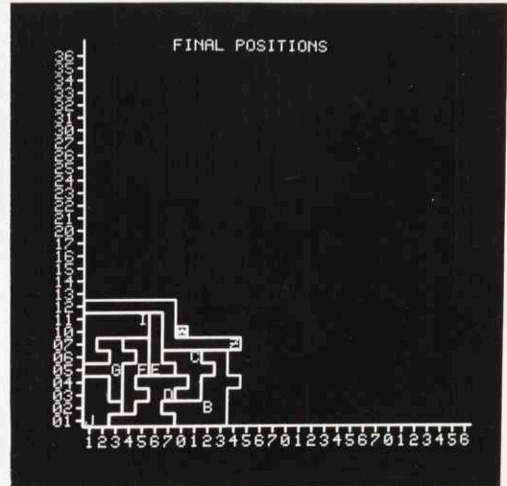

FRAME 0063

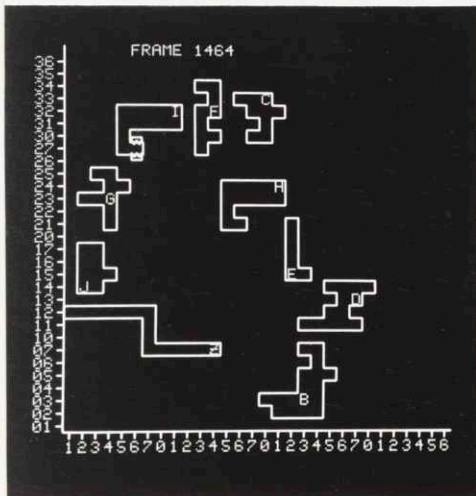

FRAME 0076

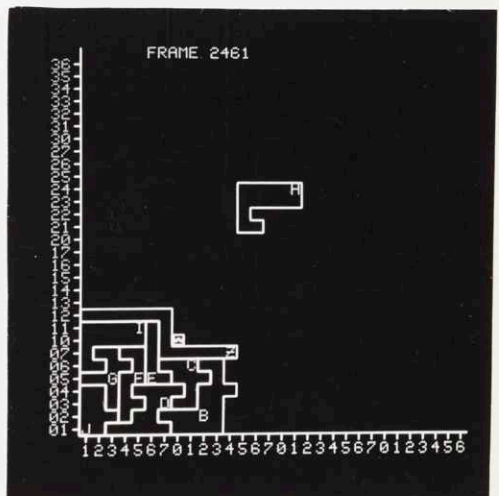
Task 6

The final task, 6, is presented to show that the system can solve an arbitrarily complex task. The goal of the task is to move object I (the object alone above the large group) to the location object H presently occupies (the upper left in the group). All objects are movable except Z, the long object. The objects in the task are B, C, D, E, G, H, I, J, and Z. Frame 1464 shows the position of the objects after they have been moved so object I can be put into place. Frame 2461 is the last frame of the task.


INITIAL POSITIONS


FINAL POSITIONS


FRAME 1464

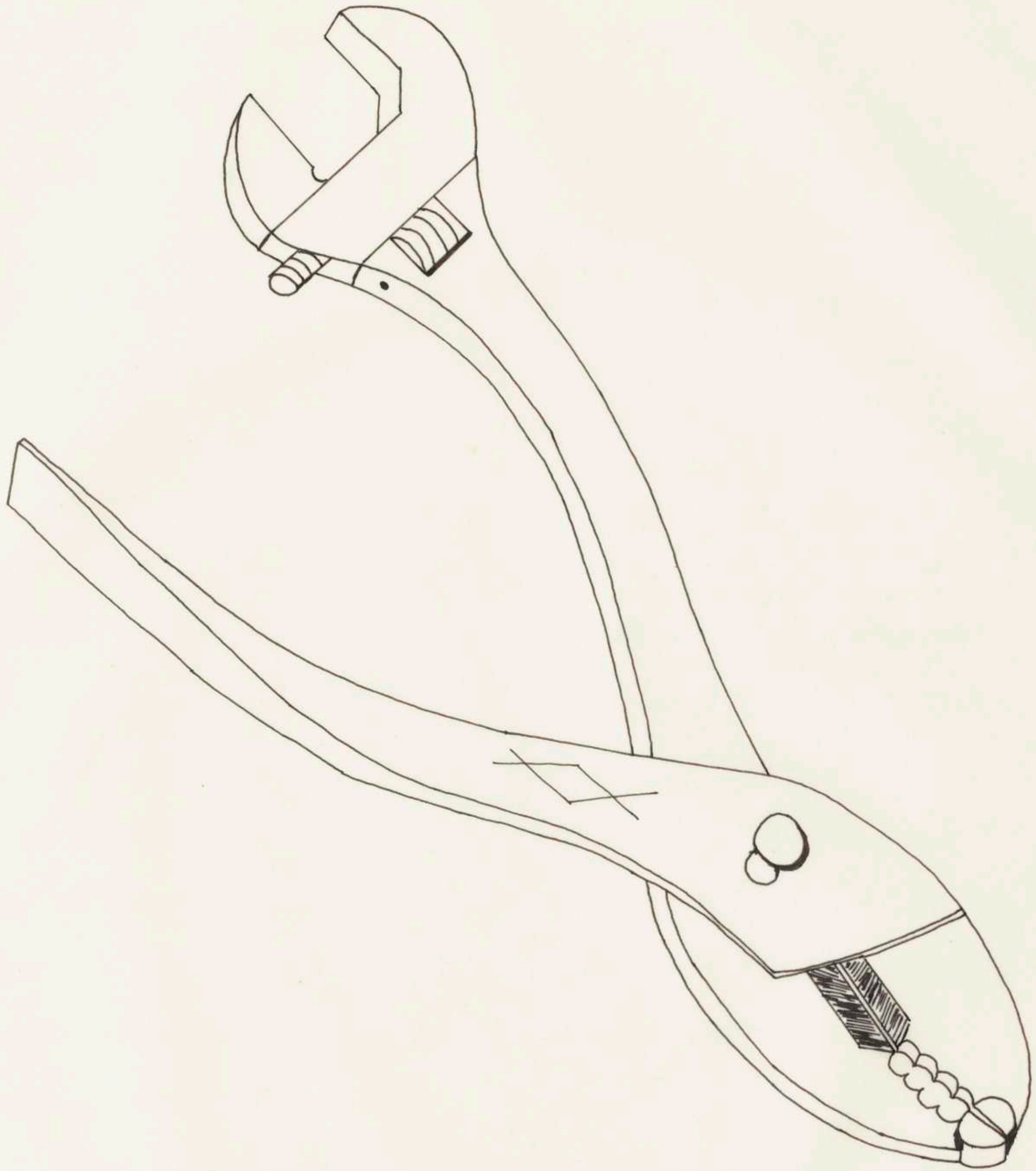

FRAME 2461

Chapter IX Specific Problems for Future Work

The work necessary to implement the system, the thought in formulating the concepts, and subsequent reflections have illuminated several problems that need future work.

What is a Handle?

The first problem is the identification of a handle on an object. For some tasks, with some objects, people have no problem distinguishing which part of an object is the handle. For example, the handle of an ordinary screwdriver is easily distinguished. But, as a counter example, consider the combination pliers, shown in figure 48, which can be used as pliers, screwdriver, wrench, wire cutter, and possibly as a hammer. One must know how he is going to use this tool before he decides which part will be the "handle." However, people normally do not consciously calculate their actions. Ordinarily, they do not think what part of an object is to be the handle. They just pick the object up by the most convenient part; and as they work, they frequently change their grip to others that better suit their immediate purposes. The process is more reflex in



Combination Pliers

Figure 48

adults than conscious effort. People, then, solve the handle problem by:

1) having the ability to change their grip frequently and easily, and
2) having the ability to know what they are going to do with an object before they pick it up.

When pushing, the problem of what a "handle" is, where to push, is a bit simpler. To push an object, one contacts the object so that the line drawn from the point where he is pushing through the center of gravity (or the center of adhesion to the support surface) of the object is in the direction the object is to move. If this position on the object is not available (for example, if the object is next to a wall), one tries to find another place to push so the object will move approximately in the desired direction.

The system uses some arbitrary rules to help it solve the problem of where an object should be grasped. The system tries to find a point closest to the middle (Y direction only) of an object to grasp. If it has a choice of places, it picks the one that is the cheapest for the jaws to get to.

In the examples run on the system, these two heuristics worked very well. (In no case was a task or object designed so as to insure the system would behave "nicely.") As an example, consider the object shown in figure 49. If the jaws can, they will grasp the right protrusion rather than either of the left ones. The decision this system makes about a place to push is very simple; it chooses the cheapest one it can find (that will move the object in the correct direction). It makes no effort to push through the center of gravity, as all motion is restricted to the X and Y directions.

As the system is presently implemented, it remembers only one grasp position and four push positions (one for each direction). The system could be made much more flexible if it remembered (and calculated paths for) all possible grasp and push positions. This procedure would give the system the ability to change from one grasp position to another if it wanted. This procedure could be implemented only by increasing the number of values on the G (grasp and push) axis in the object state space, the number of points in the space, and the processing time needed to find a shortest path. For complex objects, these increases could be a factor of ten or more, making the task almost impossible to solve using the present techniques.

Jaws will prefer to grasp the right protrusion of the object

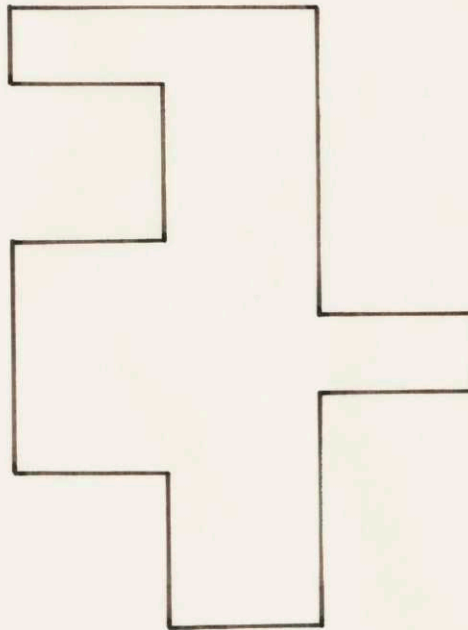


Figure 49

What is needed, then, is a method which will allow the system to change freely from one grasp position to another but will not increase the size of the state space nor increase the time necessary to compute shortest paths.

Variable Quantization of a Space

This system divides the space into equal sized squares. This procedure is straightforward to implement and gives easily interpreted solutions (paths). But it is wasteful, as a lot of unnecessary points are stored. The system allots the same amount of storage space whether a space is empty or full of objects. Also, paths across empty space are straight lines; the system should not have to compute these paths in the same way as it computes the paths around objects.

One possibility is to compute straight line paths to the corners of objects, as is done by the SRI group.[21, 24] This method assumes a path will be a straight line from start to finish, or will be straight line segments from the start, to object corners, to the finish. Figure 50 shows a path found using this method. However, the discussion in their reports[25] indicates that there is some difficulty in computing paths using these corner points.

Path found using straight line segments to the corners of objects

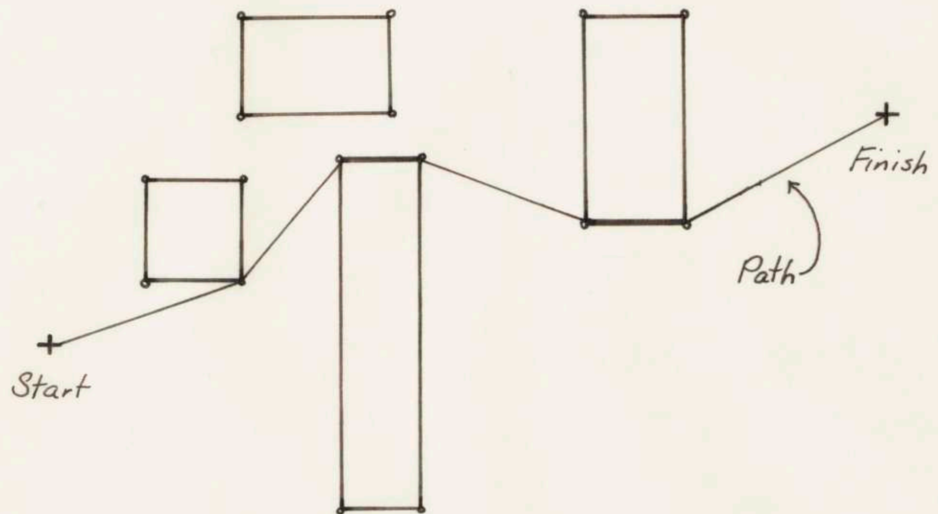


Figure 50

Another approach to the problem might be to designate points only around objects, and compute the paths in the immediate vicinity of the objects in the same way as is presently done. Paths across the empty space would be computed as straight lines. A similar approach to the problem would be to have variably spaced points in the space. The spacing would be dependent on some function of "interest" the system has in an area.

Successful implementation of a method similar to those described above would be of benefit in finding solutions to manipulation tasks. Other fields that rely on shortest path algorithms would also benefit.

Not Enough Out of the Way Places

In some cases, this system will fail to find task solutions because there are not enough Out of the Way Places. In these cases, solutions might be found if the requirements for Out of the Way Places were relaxed. As mentioned earlier, the requirements are overly restrictive. The unnecessary failures, caused by too few Out of the Way Places in the task space, occur for two reasons:

1) Objects moved to Temporary Locations are, in many cases, moved further than is necessary

to solve tasks.

2) There may be an interaction between an object's shape and its Out of the Way Place, which results in an object's being moved further out of the way than is absolutely necessary to solve tasks.

To explain reason 2, task 2 of Chapter VIII will be used as an example. Note the path planned for object A, frames 11 to 44. (This path is chosen as C is one unit wide where A crosses it, and two units wide at the other possible crossing point along the X axis. The shortest path algorithms try to move objects through as little space occupied by other objects as possible.) With this planned path, C must be moved up to Y=14. Suppose instead that A were moved down to the X axis and then left to its final position. Then C would have to be moved up only to Y=12. Here C's shape has influenced A's path. And A's path sets the requirements for C's Out of the Way Place. Hence, C's shape has influenced its Out of the Way Place.

Now to examine the two reasons for unnecessary failures cited above. Objects moved out of the way are moved to Temporary Locations so they can be moved later. However, Temporary

Locations take more than the minimum amount of space to store an object, decreasing the number of Out of the Way Places in the task space. As an object does not always have to be moved to a Temporary Location to guarantee that it can be moved later, what is needed is a method to find a location from which an object can be moved, that requires only a minimal amount of space to store the object. Discovering such a method is left as a future project.

In moving an object to a Temporary Location, the assumption has been made that the object must be moved again. However, in some cases, the object will not need to be moved again. But to keep the system from making irreversible decisions, the system can decide to move an object only to a location from which it can be moved.

Reason 2 (object's shape influences its Out of the Way Place) leads to more specific suggestions. The first suggestion is to change the algorithm to charge only once for moving one object through the space occupied by another. This procedure, unfortunately, would increase the running time of the program as partial sub-paths would have to be retraced frequently.

Another solution might be to move the in the way object to various test positions, then try computing an object's

path to determine which of the test positions is the best Out of the Way Place. This approach, however, would require computing multiple paths before some objects are moved. Hopefully, there are other ways to find "minimal" Out of the Way Places.

There are puzzles which are difficult just because there are few Out of the Way Places. A common example is the "15 puzzle." (A drawing is shown in figure 51.) There has been some study of mechanical solution of various 15 puzzle problems. Most of these have depended on sub-goal tree searching techniques, which are in essence the same as the solution technique proposed in the previous paragraph.

Pushing or Carrying More Than One Object

Giving the system the ability to carry or push more than one object presents a powerful tool for solutions to very difficult tasks. To implement this ability would require a drastic change in the state space models of the task spaces, as methods must be found which allow the state space to describe the task; but, at the same time, keep the state spaces from becoming too large. Perhaps the spaces could be segmented in a way similar to that used to segment the jaw/object spaces. But as the criteria for this new problem are

15 PUZZLE

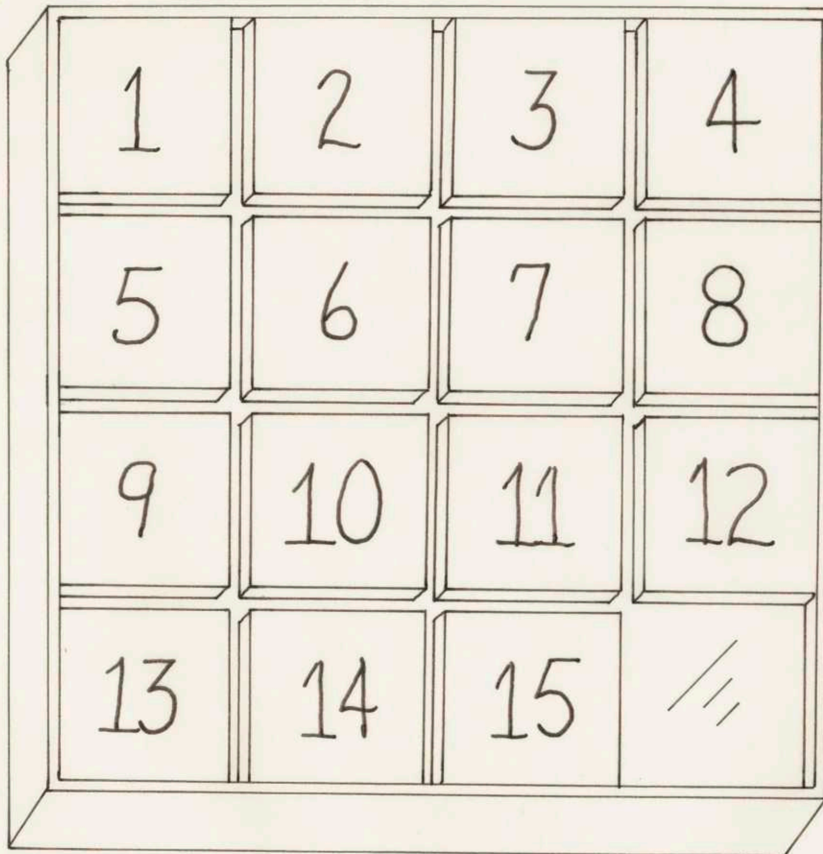


Figure 51

not as strict and straightforward, the segmentation rules will probably be more complex.

The above problem is very similar to the problem of giving the system the ability to construct sub-assemblies which would be used to build a complete assembly. There are two different ways to incorporate this ability into the system. The first is for the operator to indicate that at a particular point in a task, a particular sub-assembly must be constructed. This method could be implemented on this system as it now exists by adding the necessary functions to the TASK TREE section of the system.

The other way is to make the system figure out when the sub-assemblies must be constructed, and which objects are to be combined to form the sub-assemblies. Suppose the system is given a problem like "Here are n objects to be put together to make final form Q." If final form Q could be built from the n objects only by construction of sub-assemblies using <u>some</u> of the n objects (the others being used individually in the final construction), the system would have a combinatorial problem larger than n!. If n is much larger than 5 or so, presumably some heuristic methods must be utilized to keep the problem of manageable size.

Finding Connected Out of the Way Places

In future research, some method should be found to determine the locations that are Connected Out of the Way Places. A straight forward method to do this would be to put objects in required positions (see Chapter V for the definition of Connected Out of the Way Places), and then determine if paths can be found to (or from) various locations. But this method would require the computation of many paths, a time consuming procedure. Hopefully, a method can be found which is more practical.

Use of an N Level Method to Find Problem Solutions

To find solutions to complex manipulation tasks, the system described in this thesis uses a two level method in which the upper level deals only with abstract task requests. The philosophy of using a two level method should be extended to N levels, in which progressively higher levels deal with more highly abstracted task requests. Such a solution technique may form a useful framework for solving problems such as those that require planning activities.

Chapter X Conclusions

1. The work of this thesis has demonstrated that a two level system as specified below is a practical way of solving manipulation problems. The upper level part of the system is an AND TREE which orders sub-tasks, so that their concatenated solutions result in the solution of a specified complex task. The lower level part of the system consists of

a) a procedure for setting up a state space which describes a sub-task, and

b) a shortest path algorithm which finds the solution to the sub-task.

This scheme avoids the need for very large, many dimensional, state space searches substituting instead an ordered series of searches in smaller state spaces. It therefore makes more efficient use of computer memory than schemes previously available.

2. An operator can effectively control this system as a supervisor. He inputs a task request by specifying the desired goal state of the task space (he specifies _what_ he wants done; he lets the system figure out _how_ to accomplish it), and monitors the results. His decisions on what the

next task is, is influenced (but not determined) by the response of the system.

3. The upper level system's mode of control of the lower system is supervisory; the upper level system gives the lower level system requests and then waits to see what the results are. The lower level system's response (i.e., no solution, conditional solution, or solution) influences the next request made by the upper level system. There is no reason why one upper level system could not supervise several lower level systems. Current computer programming methods (re-entrant programming) makes this a practical possibility.

4. The AND TREE is a general data structure which can be used to order the sub-activities of any task. It performs the same functions as a PERT chart, and offers the same opportunity for general application.

Appendix A

Example of Two Stack Diamond Algorithm Finding Paths in a Space

The space is as initially shown in figure A-1, with the transition costs next to the node links. All costs are symmetrical, cost $\left[(1,1) \longrightarrow (1,2)\right]$ = cost $\left[(1,2) \longrightarrow (1,1)\right]$ =1. The starting point is at (1,1). Its cost is zero, and it is the only point on the "full" list. The other list is empty. The situation is as shown in figure A-1 and below.

FFL (Full Front List)	EFL (Empty Front List)	POINTS	COST (To get here)	BPTHCF (Best Place to have Come From)
1,1	empty	1,1	0	Start
		1,2	∞	NW (Nowhere)
		1,3	∞	NW
		2,1	∞	NW
		2,2	∞	NW
		2,3	∞	NW
		3,1	∞	NW
		3,2	∞	NW
		3,3	∞	NW

We now take all the points off the FFL. As we take each point off, we proceed as directed by the algorithm (figure 27, Chapter III). We call this procedure a front motion, as the front has moved one unit. The situation is as shown below, and in figure A-2. Note that only arrows marked #1 are in place now.

1. First Front Motion

FFL	EFL	POINTS	COSTS	BPTHCF	
	1,2	1,1	0	Start	
	2,1	1,2	1	1,1	———
		1,3	∞	NW	
		2,1	9	1,1	———
		2,2	∞	NW	
		2,3	∞	NW	
		3,1	∞	NW	
		3,2	∞	NW	
		3,3	∞	NW	

Note: Changes are indicated by a bold line to the right.

We swap the two lists and then proceed until both lists are empty, which indicates the algorithm has terminated.

2. Second Front Motion

FFL	EFL	POINTS	COSTS	BPTHCF	
1,2	3,1	1,1	0	Start	
2,1	2,2	1,2	1	1,1	
	1,3	1,3	2	1,2	—
		2,1	9	1,1	
		2,2	9	1,2	—
		2,3	∞	NW	
		3,1	10	2,1	—
		3,2	∞	NW	
		3,3	∞	NW	

3. Third Front Motion

FFL	EFL	POINTS	COSTS	BPTHCF	
3,1 (a)	2,3 (c)	1,1	0	Start	
2,2 (b)	2,3 (b)	1,2	1	1,1	
1,3 (c)	3,2 (a)	1,3	2	1,2	
		2,1	9	1,1	
		2,2	9	1,2	
		2,3	{15 (b) 3 (c)	{2,2 (b) 1,3 (c)	——
		3,1	10	2,1	
		3,2	11	3,1	——
		3,3	∞	NW	

Note that the point 2,3 was put on the EFL twice, as the result of the order in which the algorithm computed paths and costs.

4. Fourth Front Motion

FFL	EFL	POINTS	COSTS	BPTHCF	
2,3	3,3	1,1	0	Start	
2,3		1,2	1	1,1	
3,2		1,3	2	1,2	
		2,1	9	1,1	
		2,2	9	1,2	
		2,3	3	1,3	
		3,1	10	2,1	
		3,2	11	3,1	
		3,3	4	2,3	——

5. Fifth Front Motion

FFL	EFL	POINTS	COSTS	BPTHCF	
3,3	3,2	1,1	0	Start	
		1,2	1	1,1	
		1,3	2	1,2	
		2,1	9	1,1	
		2,2	9	1,2	
		2,3	3	1,3	
See figure A-3 for		3,1	10	2,1	
changes in the path.		3,2	5	3,3	——
		3,3	4	2,3	

6. Sixth Front Motion

FFL	EFL	POINTS	COSTS	BPTHCF	
3,2	3,1	1,1	0	Start	
		1,2	1	1,1	
		1,3	2	1,2	
		2,1	9	1,1	
		2,2	9	1,2	
		2,3	3	1,3	
		3,1	6	3,2	——
		3,2	5	3,3	
		3,3	4	2,3	

7. Seventh Front Motion

FFL	EFL	POINTS	COSTS	BPTHCF	
3,1	2,1	1,1	0	Start	
		1,2	1	1,1	
		1,3	2	1,2	
		2,1	7	3,1	——
		2,2	9	1,2	
		2,3	3	1,3	
		3,1	6	3,2	
		3,2	5	3,3	
		3,3	4	2,3	

8. Eighth Front Motion

FFL	EFL	POINTS	COSTS	BPTHCF
2,1	2,2	1,1	0	Start
		1,2	1	1,1
		1,3	2	1,2
		2,1	7	3,1
		2,2	8	2,1
		2,3	3	1,3
		3,1	6	3,2
		3,2	5	3,3
		3,3	4	2,3

9. Ninth Front Motion

FFL	EFL
2,2	

No changes are made in costs, and no point is put on the EFL. The algorithm terminates with all paths and costs as found during Front Motion Eight. The paths from any point to the start are as shown in figure A-5. Just trace back as directed by the arrows.

Initial configuration, no paths

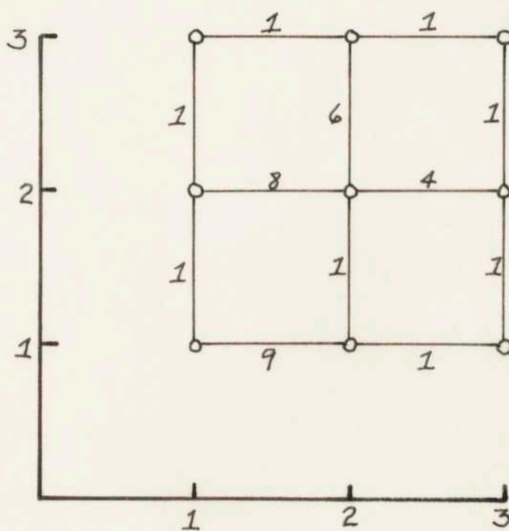


Figure A-1

Initial paths

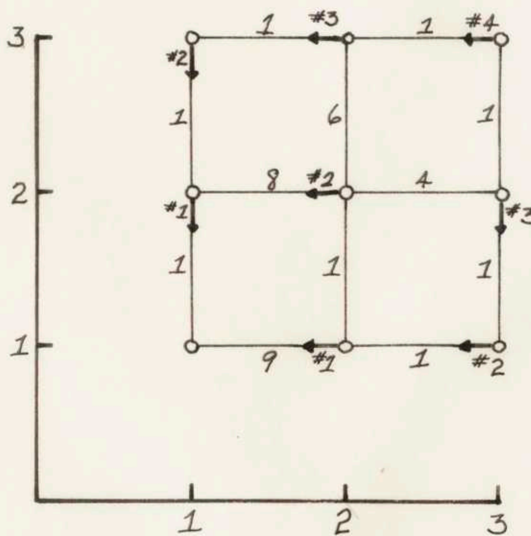


Figure A-2

Intermediate paths - Changes are numbered

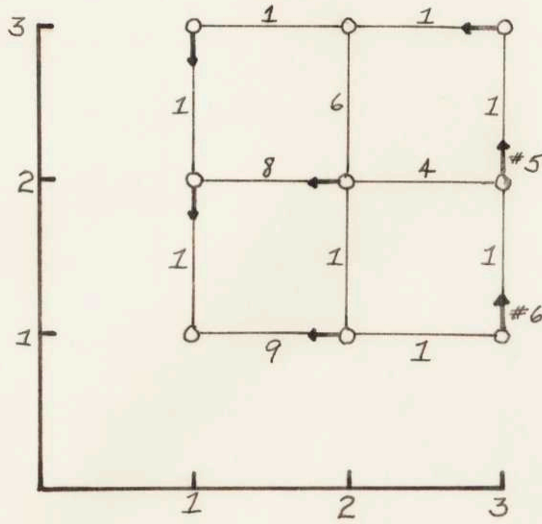


Figure A-3

Final paths

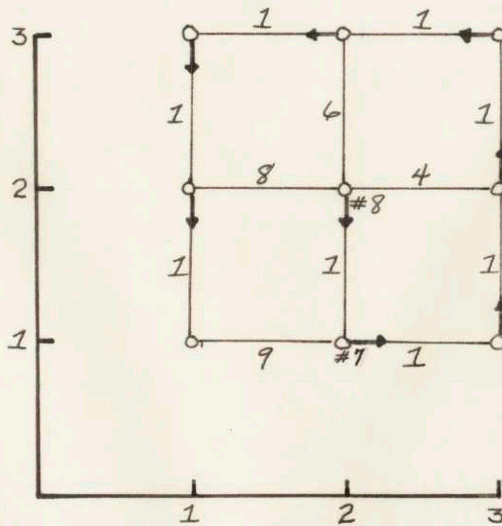


Figure A-4

Appendix B

What to do when an object at its specified final position occupies part of the same space as a movable object (that has no specified final position) at its present position.

The above problem can be encountered when finding the order to move objects to their final positions. Figure B-1 shows a task in which this occurs. Figure B-1-a shows the initial positions and figure B-1-b the final positions. Figure B-1-c shows the position of objects when the system starts to find the order to move the objects to their final positions.

The system, as described earlier, attempts to solve the problem in the following way. In figure B-1-c it plans to move A and discovers that C is in the way. It then tries to move C out of the way, and discovers that A is in the way. It tries to move A out of the way and discovers C is in the way, still. The system then discovers the loop and eventually finds the task impossible.

To prevent the above failure, the following amendment must be made in the system procedure. When the system plans to move an object out of the way that has no specified final position, and finds that a second object is in the planned path, the system must check to determine if the second object

Initial Positions

Final Positions

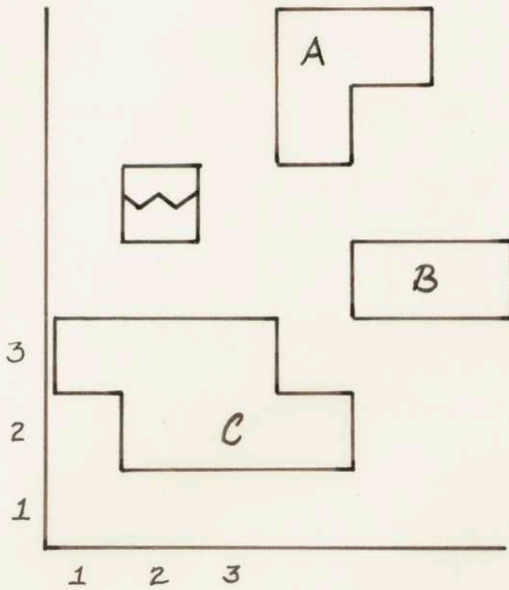


Figure B-1-a

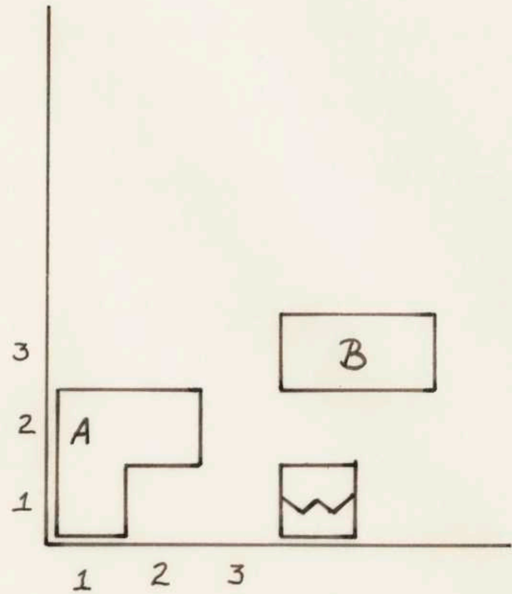


Figure B-1-b

Overlap problem as described in Appendix B.

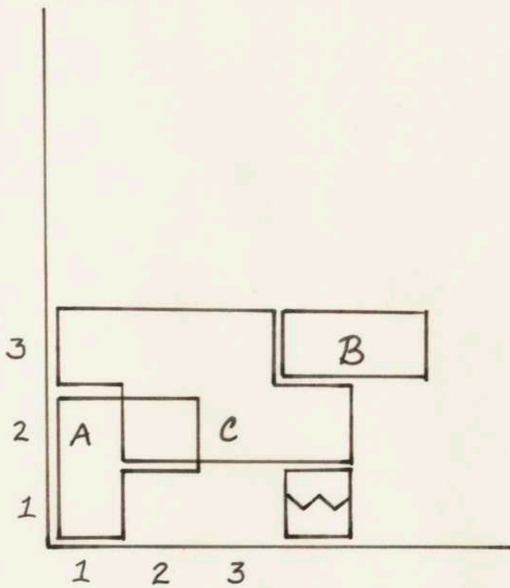


Figure B-1-c

Position of objects when system starts to find order to move objects to their Final Positions.

Objects A and C overlap at (2,2).

is in the part of the path the object occupies when it is at its initial position. If this is true, and if this other object has a specified final position, then the object is moved to an Out of the Way Place, and the second object is ignored.

The only circumstance where there can be overlapping objects is when the system is determining the order to move objects to their final positions. There exists the possibility that an overlap might be specified in the input of the initial or final positions, but a well-designed input routine would prevent this.

Appendix C

Program Documentation

There are five programs that compose the system. Four of the programs make up a complete system. The fifth is a stand alone program which can perform all of the input/output functions of the systems and requires much less core than the complete system. All the programs can be run only on the M.I.T. Project MAC Artificial Intelligence Group's PDP-10 Time Sharing System,[6] circa September, 1970, or other computer systems that are *hardware* and *software* compatible.

The four programs that compose a complete system are retained in ASCII code to facilitate changes. These programs are

INS 34

TREE 20

OBJSMS 31

CSP 5

as of September 15, 1970. The second names of the programs, the numbers, are subject to change as improvements are made. Each program should be assembled and the binary files loaded using the *linking* loader. The program is started at location

"START". The system will reply with several line feeds followed by the word "READY". The program is now ready for input, to be described below. The loading and starting procedures are not described in detail as software improvements are rapidly being implemented and any descriptions given here will probably be quickly outdated. For specific loading and starting instructions, it is suggested that a person familiar with the computer operating system be contacted.

The input/output program is INA 25. It is assembled and started the same as the complete system, but it can be loaded using the regular loader.

Input/Output Format

The input for the system can be from three sources: teletype, disk, or micro-tape. When started, the program looks for input from the teletype.

The control character for the input is the colon (:). It preceeds the letter that designates what is to be input. The input functions are

:T Allows one to change from one input source to another; e.g., disk to teletype.

:O For input of objects' descriptions at initial position.

:I To change objects' initial positions and to specify the jaw's initial position.

:F To specify objects' and jaw's final positions.

:D To delete an object from the task space.

:S To check initial and final configurations of objects.

:R To read results, previously computed, that are stored in tape or disk files.

:GO To start the system to find a solution to the task.

:S, :R, and :GO can be given only from teletype control.

<u>:T</u>

The operator has the choice of three inputs.

TTY↲ (↲ represents carriage return.)

For input from teletype.

DSK:flnml flnm2↲ ("flnml" is short for "file name 1".)

For input from flnml flnm2 on disk.

UTn:flnml flnm2↲

For input from flnml flnm2 on a micro-tape mounted on unit <u>n</u>.

If an input file ROPE 6 is to be read off a micro-tape mounted on unit 3, the request would be

```
:T
UT3:ROPE 6↲
```

When input is complete, a message is printed on the console. The last characters in ROPE 6 must be

```
:T
TTY↲
```

to return control of input to teletype.

The system supplies some carriage returns, others must be supplied by the operator. The best rule is to type a line, wait a second, and if no carriage return is generated, type it.

:0

The format here is of two types. The first is

name,M or F,(n1, n2) OR ,n3) ("~~" delimits OR choice.)

Here "name" is the name of an object to be described. "name" can be any combination of six letters and numbers. As the left character is used to identify the object in the various displays, most objects receive one letter names.

"M or F", either letter may be used. "M" specifies the object is movable; "F" specifies the object is fixed, immovable.

n1 is the X coordinate of the base location.

n2 is the Y coordinate of the base location.

n3 is the cost of moving through the space occupied by this object.

n3 is supplied if the operator wishes to supply this cost versus having the system compute it. A cost less than or equal to zero is not allowed.

The commas and parentheses must be supplied as field delimiters.

The second format of objects' descriptions is

n1,n2

n1 is the X coordinate of a location occupied by the object.

n2 is the Y coordinate of a location occupied by the object.

There are usually several lines in this second format. The object's base location must be included as one entry. The list is terminal by a line "0,0)".

For example, the description set

```
:O

KU2,M,(10,15),5
10,15
10,16
10,17
11,17
0,0
```

defines object KU2, that is movable, and has its base location at (10,15). It will cost another object 5 units to move through the space occupied by KU2. See figure C-1.

There are various error detecting mechanisms in the input routines, but they cannot be guaranteed to catch all errors. The only way to be sure the system has understood what is meant is to check initial and final positions with the :S command.

:I

The format here is

name,(n1,n2)) OR ,n3)

Again,

"name" is the name of the object

Video display of object KU2

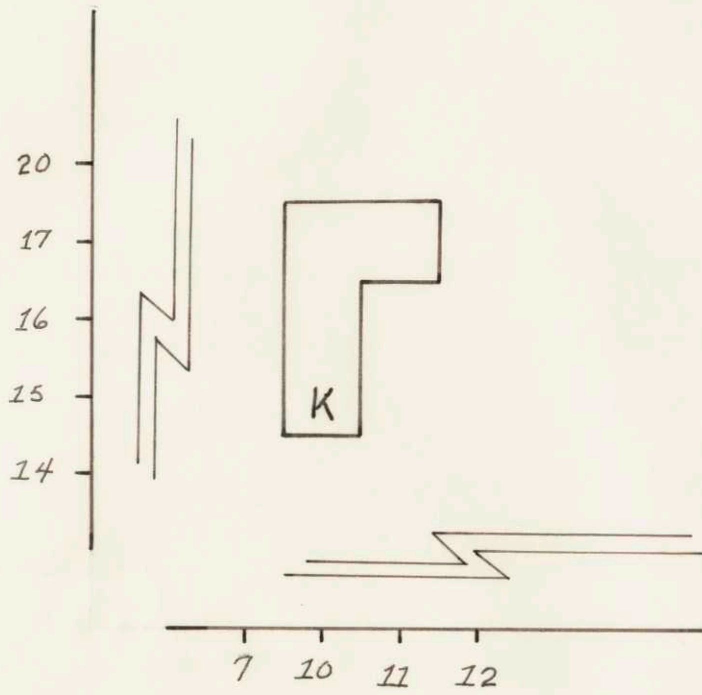


Figure C-1

n1 is the X coordinate of the new base location

n2 is the Y coordinate of the new base location

n3 is the same as described under the ":O" command.

If the name is "JAWS", this command sets the initial position of the jaws.

<u>:F</u>

The format here is

name,(n1,n2)

(n1,n2) is the desired base location of the object when the task has been completed. If (n1,n2) is equal to (0,0), the specified final position of object "name" is removed. If the input line is "KILALL", all specified final locations are erased.

<u>:D</u>

The format here is

name

The object "name" is deleted from the task space. If name is "KILALL", all objects are deleted from the space.

<u>:S</u>

There are several options available here.

T↲

If the next line is "T ", an abbreviated description of the task site's initial and final positions are printed on the teletype.

L I↲

This command causes a picture of the Initial Position of the object in the task space to be printed on the line printer.

L F↲

This command causes a picture of the Final Position of the objects in the task space to be printed on the line printer.

V I↲

This causes the video display to show the initial position of the objects. Typing a "K" turns the display off and allows the program to proceed.

V F↲

This causes the video display to show the Final Positions of objects. Typing a "K" turns the display off and allows the program to proceed.

Error returns will be generated if the program cannot "sieze" the line printer or video display as necessary.

<u>:R</u>

The system responds with

FROM-

The input format is

DSK:flnml flnm2

to read disk file "flnml flnm2", or

UTn:flnml flnm2

to read file "flnml flnm2" from micro-tape mounted on drive n.

The file "flnml flnm2" is the result of previous computations of the system, saved after the system completed a task solution.

The response of the system is a few carriage returns followed by the line

MOTION DISPLAY

The previously computed path can be viewed. There are three forms of output: the video display, the line printer, and teletype. The format for these displays is

V A↲

This starts the video display at the start of the task. The solution can be seen by hitting the space bar once for every frame. To reverse the motion, type an R for each frame. To have the motion automatically displayed, type an A . The automatic display can be stopped at any time by typing any character. Then spaces, R's, or another A can be typed to continue.

To start the video display at the last sub-task executed, type

V L↲

followed by the desired string of spaces, R's, or A's.

The display will run until the last frame of the task motion is shown, where it will hang with no response to A's, R's, or spaces. Typing a K will turn off the display. The system is now ready for the next command to the motion display routine.

A sample interaction with the display routine is

V A↲
RRARRRR RA K

Caution: if the first display command after the computer types MOTION DISPLAY is "V L ", the display will show very strange results.

Before one gets output from the line printer, it is expected that the video display will have been seen, and particularly desirable frames selected for permanent copies.

The format for line printer output is

L↲
n1↲
n2↲
n3↲
.
.
.
0↲

L signifies the line printer output is desired. n1, n2, and n3 are frames to be printed. The last entry, 0 (the number) denotes the end of the list.

A sample printout from the printer is shown in figure C-2. This is frame 1464 of task 6 in Chapter VIII. The first character of the object's name shows the locations the object occupies. The * denotes the base location. An object that occupies only one location would be represented by a *. The interpretation of JK, JX, etc., is given in the description of the teletype output, below. The jaws are represented by the =, one = for each half. If the jaws are closed, only one = is shown.

The teletype output is the contents of the list the system keeps to define the task solution, the description of the jaw's and objects' paths.

There are two options for teletype output. The first is

T L↲

This outputs the path that is the solution of the last sub-task the system solved. The first line of output is

Example of line printer output

```
FRAME  1464 .

OBJECT BEING MOVED IS   I .

PARAMETERS ARE   JK=02, JX=06, JY=26,, OG=00, OX=11, OY=32 .

  + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
36+                                                         +
35+                                                         +
34+                     F F                                 +
33+                       F   C C *                         +
32+         I I I I *   F *     C C C                       +
31+         I I I I I   F         C                         +
30+         I =         F F     C C                         +
27+         I I         F                                   +
26+           =                                             +
25+     G G                                                 +
24+       G G               H H H H *                       +
23+   G G *                 H H H H H                       +
22+       G                 H                               +
21+       G                 H H       E                     +
20+                                   E                     +
17+   J J                             E                     +
16+   J J                             E                     +
15+   J J J                           * E                   +
14+   * J                                     D D D D       +
13+                                             D *         +
12+ Z Z Z Z Z Z Z                               D D         +
11+             Z                       D D D D D           +
10+             Z                                           +
07+             Z Z Z Z Z *             B B                 +
06+                                       B                 +
05+                                     B B B               +
04+                                     B B                 +
03+                               B B B * B                 +
02+                                 B B B B                 +
01+                                                         +
  + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
    0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3
    1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4
```

Figure C-2

a set of labels,

JK, JX, JY, ,, OG, OX, OY

which will be explained later.

The next line is the name of the object moved. The third line tells whether this object could be moved, or whether the solution was a plan: the line is PLAN or EXECUTION.

On the lines following this are six two digit numbers per line separated by commas:

n1, n2, n3, ,, n4, n5, n6

n1 is directly below JK and tells how far open the jaws are. n1=00 means the jaws are closed.

n2 is below JX and is the X coordinate of the lower half of the jaws.

n3 comes below JY, and is the Y coordinate of the lower half of the jaws.

n4 is below OG, and tells how the object is being moved.

If OG=10, the jaws are not in contact with the object.

If OG=00, the object is being grasped.

If OG=01, the jaws are in contact with the object in a position to push it in the -X direction.

If OG=02, the jaws are in contact with the object in a position to push it in the +X direction.

If OG=03, the jaws are in contact with the object in a position to push it in the -Y direction.

If OG=04, the jaws are in contact with the object in a position to push it in the +Y direction.

n5 is directly below OX, and defines the object's base location X coordinate.

n6 is below OY, and defines the Y coordinate of the object's base location.

The other format alternative for teletype output is

```
T F↲
n1↲
n2↲
```

which designates that the path list between frames n1 and n2 will be typed out in the format described above. n1 must be less than n2.

Requests for output from the teletype, line printer, and video display may be made in any order.

To escape from the motion display, type an N where new output would normally be requested.

:GO

This command starts the program running to find a solution to the task. If it is given to the input/output

program (INA) an error return is generated.

As the system computes each sub-task, the system types out

MOTION DISPLAY

if the program has control of the teletype. If the system does not have control of the teletype, no output is attempted.

The operator's responses here are the same as described under :R except that typing N↲ to escape produces the question

PRINT OUT TREE?

A response of YES↲ prints out the TASK TREE on the line printer as it was just before execution of the last sub-task was attempted. After the TASK TREE is printed, the system continues to complete the solution of the task. If any other response to the question is given, the TASK TREE is not printed.

When the system completes the solution of a complete task,

THEEND ≫ .VALUE

is printed at the teletype. When the program is "proceeded" (ask for help if not familiar with the procedure) the system will ask

SAVE RESULTS?

A response of YES↲ will cause the system to type

ON-

The response is device:flnml flnm2 to save the path on "device" as "flnml flnm2." Usual device names are DSK (disk) or UTn (micro-tape unit n). The information is saved in image mode; i.e., the binary contents of memory locations. The saved results can be read using the :R command.

After the results have been saved, the program is restarted automatically, but with objects left in their final positions. Type

```
:T
TTY

:S
V I
```

to see the initial positions as they are now.

The easiest way to input initial conditions into the system is to type a file with objects' descriptions and final positions using the editor program, TECO. The last characters of the file must be

```
:T↲
TTY↲
```

to return control of the input program to the teletype.

Appendix D

Flow Chart of the System

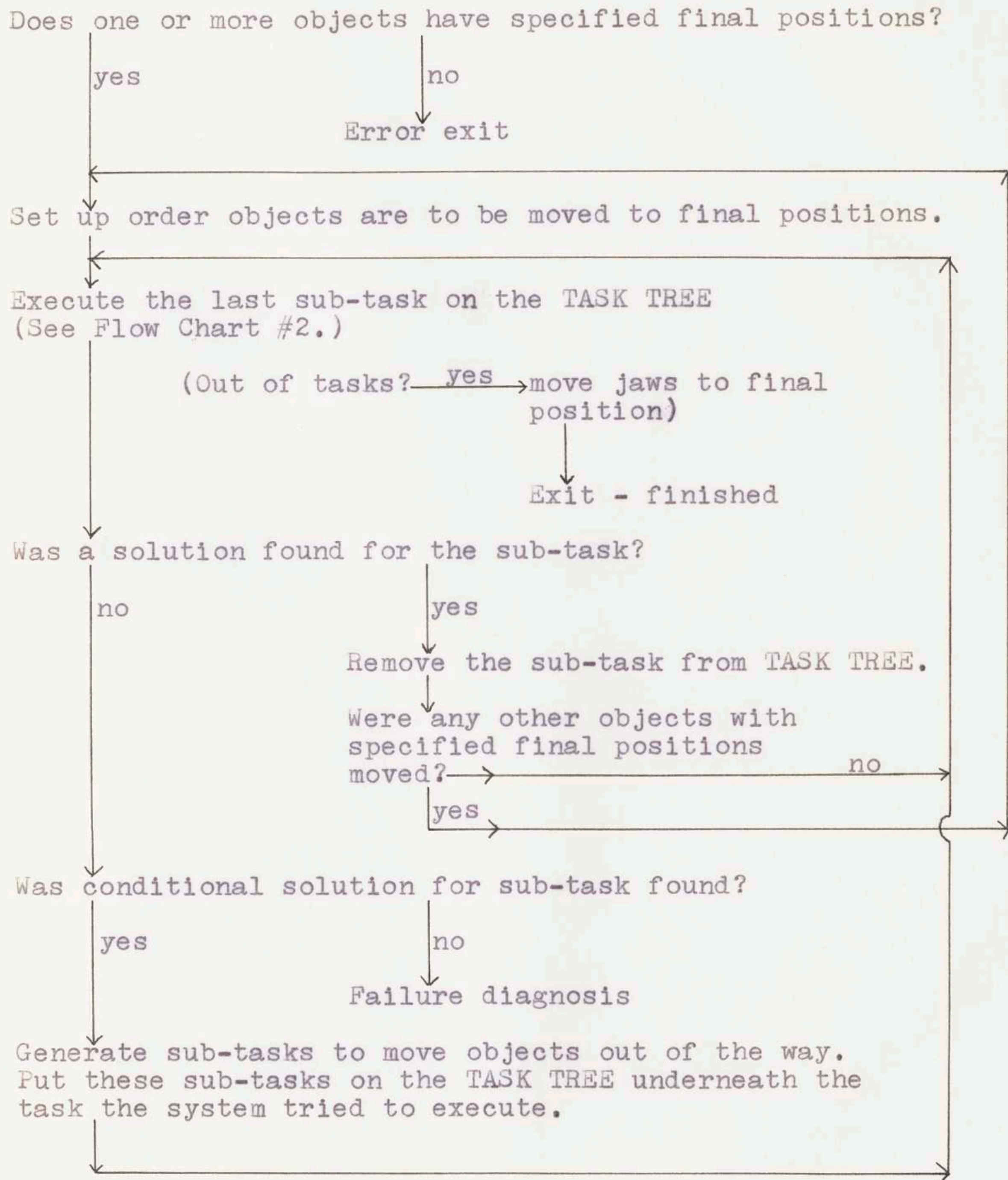
There will be no flow chart of the input/output for the system as the functions are described in the sections on formats. The input/output section of the program constructs three lists which it gives to the next part of the program. These lists are:

1) The Active Object List which contains objects' descriptions and objects' positions in the space.
2) The Abstracted Object List which contains objects' descriptions coded as though the base location were at (0,0).
3) The Finishing Position List which contains the names and final positions of those objects that have specified final positions.

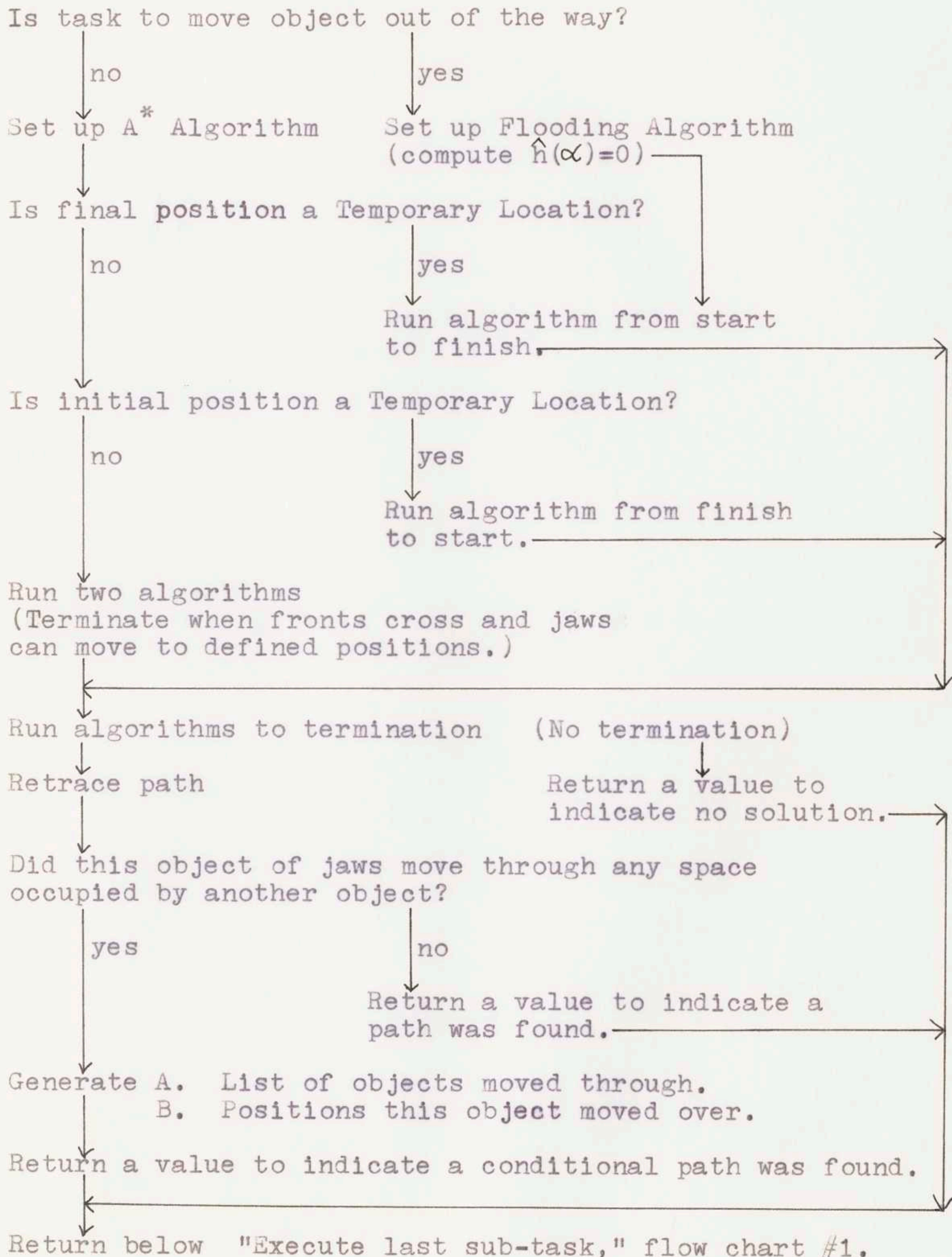
The system flow chart is high level, showing how the various components interact. Detailed information of functions is included in the text of this thesis.

SYSTEM FLOW CHARTS

Flow Chart #1



Flow Chart #2 Execute Last Sub-Task



References

1. Archibald, R. D., and R. L. Villoria. Network-based Management System (PERT/CPM). New York: John Wiley & Sons, 1967.

2. Barber, D. J., MANTRAN: A Symbolic Language for Supervisory Control of an Intelligent Remote Manipulator. S.M. Thesis (M.E.), M.I.T., June, 1967.

3. Berge, Claude. The Theory of Graphs and its Applications. Translated by Alison Doig. New York: John Wiley & Sons, 1964.

4. Deo, Narsingh. An Extensive English Language Bibliography on Graph Theory and Its Applications. NASA Technical Report 32-1413, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California. October, 1969.

5. Digital Equipment Corporation. PDP-8 User's Handbook. Maynard, Massachusetts.

6. Eastlake, Donald E. "ITS 1.5 Reference Manual." M.I.T. Project MAC Artificial Intelligence memo #161A. Cambridge, Massachusetts. July, 1969.

7. Ferrell, W. R. Remote Manipulation with Transmission Delay. Ph.D. Thesis (M.E.), M.I.T., September, 1964.

8. Ferrell, W. R. and T. B. Sheridan. "Supervisory Control of Remote Manipulation." IEEE Spectrum, Vol. 4, No. 10, (October, 1967) pp. 81f.

9. Goertz, R. C. "Manipulators Used for Handling Radioactive Materials." Chapter 27 of Human Factors in Technology. New York: McGraw-Hill, 1963.

10. Green, C. "Theorem Proving by Resolution as a Basis For Question-Answering Systems." Machine Intelligence 4, D. Michie and B. Meltzer, eds., Edinburgh University Press, Edinburgh, Scotland, 1969.

11. Hart, P. E., N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." IEEE Transactions on Systems Science and Cybernetics, Vol. 4, No. 2, (July, 1968) pp. 100f.

12. Hewitt, C. "PLANNER: A Language for Proving Theorems in Robots." in Proceedings of the Joint Conference on Artificial Intelligence. Washington, D. C. May 7-9, 1969.

13. Johnsen, E. G. "The Case for Localized Control Loops for Remote Manipulators." Paper presented at IEEE Human Factors Group Symposium, Boston, Massachusetts. May, 1965.

14. Johnsen, E. G. and Charles B. Magee, editors. Advancements in Teleoperator Systems. A colloquium held at University of Denver, Denver, Colorado. February 26-27, 1969. NASA SP-5081.

15. Johnsen, E. G. and W. R. Corliss. Teleoperator Controls. NASA SP-5070, 1968.

16. McCandlish, S. G. A Computer Simulation Experiment of Supervisory Control of Remote Manipulation. S.M. Thesis (M.E.) M.I.T. June, 1966.

17. Massachusetts Institute of Technology. Project MAC Progress Report VI. Cambridge, Massachusetts. 1969.

18. Newell, A., J. C. Shaw, and H. A. Simon. "Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics." in Computers and Thought, E. A. Feigenbaum and J. Feldman, eds., New York: McGraw-Hill, 1963.

19. Newell, A. and H. A. Simon. "GPS, A Program That Simulates Human Thought." in Feigenbaum and Feldman.

20. Nicholson, T. A. J. "Finding the Shortest Route Between Two Points in a Network." Computer J. Vol. 9, (1966) pp. 275f.

21. Nilsson, N. J. "A Mobile Automation: An Application of Artificial Intelligence Techniques." in Proceedings of the International Joint Conference on Artificial Intelligence. Washington, D. C. May 7-9, 1969.

22. Ore, Oystein. Theory of Graphs. American Mathematical Society. Providence, Rhode Island. 1967.

23. Pollack, M. and W. Wiebenson. "Solution of the Shortest-Route Problem--A Review." Operations Research, Vol. 8, (1960) pp. 224f.

24. Rosen, Charles A. and Nils J. Nilsson, eds. Application of Intelligent Automata to Reconnaissance. Third Interim Report, 18 March to 17 December, 1967. Prepared for Rome Air Development Center, Stanford Research Institute, Menlo Park, California.

25. Rosen, Charles A. and Nils J. Nilsson, eds. Application of Intelligent Automata to Reconnaissance. pp. 20-23.

26. Shaffer, L. R., J. B. Ritter, and W. O. Meyer, The Critical-Path Method. New York: McGraw-Hill, 1965.

27. Sheridan, T. B. "Use of Artificial Computation Loops Within Human Control Loops for Remote Manipulation." Unpublished memo, M.I.T. October 28, 1964.

28. Simmons, R. F. "Natural Language Question-Answering Systems--1969." Communications of the ACM, Vol. 13, No. 1, (January, 1970) pp. 15f.

29. Sklansky, Jack. "Recognizing Convex Blobs." in Proceedings of the Joint Conference on Artificial Intelligence. Washington, D. C. May 7-9, 1969.

30. Tou, Julius. Modern Control Theory. New York: McGraw-Hill, 1964. pp. 62-116.

31. Whitney, D. E. "State Space Models of Remote Manipulation Tasks." IEEE Transactions on Automatic Control, Vol. AC-14, No. 6, (December, 1969) pp. 617f.

32. Whitney, D. E. State Space Models of Remote Manipulation Tasks. Ph.D. Thesis (M.E.), M.I.T., January, 1968.

BIOGRAPHY

Philip A. Hardin was born in Atlanta, Georgia, on March 24, 1943, and received his elementary education in the public schools of Forsyth, Georgia. He attended high school at The Baylor School, Chattanooga, Tennessee. He entered M.I.T. in 1961, and was active in intercollege sports and other extra curricular activities including Pi Tau Sigma. He received a Bachelors Degree in June, 1965, a Masters Degree in August, 1966, and a Mechanical Engineers Degree in February, 1969, all from M.I.T. During his graduate years, he was a Research Assistant. He is a member of Sigma Xi and is a Lieutenant in the United States Naval Reserve.