STATE SPACE MODELS OF REMOTE MANIPULATION TASKS

by

Daniel Eugene Whitney

S.B. Humanities and Engineering
Massachusetts Institute of Technology, 1960

S.B. Mechanical Engineering
Massachusetts Institute of Technology, 1961

S.M. Mechanical Engineering
Massachusetts Institute of Technology, 1965

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

January, 1968

Signature of Author . . . .
Department of Mechanical Engineering, January 5, 1968

Certified by. . . . .
Thesis Supervisor

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

STATE SPACE MODELS OF REMOTE MANIPULATION TASKS

by

Daniel E. Whitney

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, January, 1968.

ABSTRACT

Remote manipulation is usually difficult even if the human operator is close to this work, because typically there is meager feedback and the apparatus is clumsy and hard to control. Add to this a significant time delay and efficient manipulation becomes almost impossible. This thesis presents a formal structure by which a computer may aid the operator and the manipulator. The computer (or computers) maintains a model of the task site, controls the manipulator, and receives commands from the operator.

The model is a discrete representation of all the consequences of executing atomic commands selected from a limited set, commands such as "Move manipulator jaws left one unit," or "Open jaws," which themselves can be preprogrammed routines. The consequences of any command or string of commands is a new configuration of objects and jaws. Each configuration, differing from its most similar neighbors by what one atomic command can accomplish, is called a state of the task site. Hence, the model is a state space representation of the task possibilities theoretically attainable using strings of these commands.

A task presumably begins with the task site occupying one such state. The operator may request any alteration in the environment whose final configuration is represented by another state. He thus is enabled to give commands such as "Put the wrench on the shelf," or "Put plug A into socket B."

Upon receiving the operator's command, the computer must find a sequence of atomic commands which, in principle, will carry the task site from the current state to the desired state. Each command is assigned a cost, which may depend on fuel or time consumed, risk or uncertainty, or arbitrary units. These costs may vary with the physical region of the task site, depending on level of risk or knowledge of the site. A search algorithm finds that path between initial and final states which costs the least. Since each leg of the path corresponds to the execution of one atomic command, the path may be read as an ordered work description to the manipulator, and comprises a plan for accomplishing the task.

A much smaller computer, located near the manipulator, can put this plan into effect, observing touch sensors and comparing progress to the plan's expectations. In case of collision or other mishap, it can direct reflex action more quickly than could the distant operator. The operator can concentrate on commands which specify goals and need not concern himself with the minute details of how these goals should be accomplished, nor with the actual execution. The operator is thus afforded a measure of control over the task site itself, not merely over the manipulator.

Chapter I relates this work to similar studies in artificial intelligence and optimal control theory. Work in both fields consists of finding "paths" through abstract spaces in one sense or another. Chapter II introduces the manipulation state space and employs finite graph theory to represent the space and organize it for algorithmic search. Six examples are given in Chapter III, showing how some non-trivial manipulation tasks can be expressed with discrete state spaces, such as pushing an object with the jaws or deciding how many and in what order should objects in the way be moved aside. Several algorithms are discussed in Chapter IV, and are related to conventional Dynamic Programming and a heuristic search procedure. Chapter V describes state space path-finding by means of sequences of small state spaces rather than one big space. The small spaces are selected by means of operator commands in the form of (possibly recursive) functions, and offer great savings in computer memory space and execution time over previously discussed methods. Apparatus used to demonstrate the ideas in Chapters II through IV is discussed in Chapter VI, while Chapter VII is a brief look into the future.

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

ACKNOWLEDGEMENTS

My most sincere thanks to my committee: Professor T.B. Sheridan, whose wealth of ideas nourishes the Man-Machine Systems Laboratory; Professor W.R. Ferrell with his sharp eye for experiment and language; and Professor L.R. Young, who found a spot for my work in his crowded schedule. All contributed with insight and encouragement. Thanks, too, to my wife, Dr. Cynthia K. Whitney, for reading the manuscript, and to Miss Pat Perry for a thoroughly excellent typing job.

The work reported in this thesis was supported by the National Aeronautics and Space Administration under Grant NsG 107-61 to the Man-Machine Systems Laboratory, Engineering Projects Laboratory, Mechanical Engineering Department of M.I.T.

TABLE OF CONTENTS

ABSTRACT 2

ACKNOWLEDGEMENTS 4

CHAPTER I. INTRODUCTION AND PROBLEM STATEMENT 7

Review of Previous Work 10

Discussion of Some Aspects of the Problem 19

Similar Work in Allied Fields 22

A Preview of the Method 25

CHAPTER II. THE STATE SPACE MODEL FOR MANIPULATION TASKS 32

Planning and Quantization 34

States and State Spaces 40

Representation of the State Space as a Finite Graph 45

The Task Plan Found Via A Shortest Path Problem . . 52

General Remarks 60

CHAPTER III. SOME EXAMPLES OF STATE SPACE MODELS OF MANIPULATION TASKS 64

CHAPTER IV. SHORTEST PATH PROBLEMS 81

Algorithms 85

Multipath Solutions 95

Efficiency of Systematic and Random Procedures . . . 99

Efficiency of Algorithmic and Heuristic Procedures . 104

CHAPTER V. EXTENDING THE POWER OF THE STATE SPACE MANIPULATION METHOD 109
A Minimal State Vector 113
Manipulation Functions 123
More Complicated Manipulation Functions 127
Recursive Manipulation Functions 131
Some Remarks on Language 138
General Remarks 140

CHAPTER VI. A PHYSICAL DEMONSTRATION OF STATE SPACE MANIPULATION CONTROL 146
What the Demonstration Taught 153
Film Record 157

CHAPTER VII. A LOOK INTO THE FUTURE 158

CONCLUSIONS 163

APPENDIX I. MATHEMATICAL STATEMENT OF GRAPH THEORY AND REMOTE MANIPULATION 166

APPENDIX II. TOUCH SENSOR DESIGN AND PERFORMANCE 171

APPENDIX III. BRIEF DESCRIPTION OF DEMONSTRATION COMPUTER PROGRAM. . 179

BIOGRAPHICAL SKETCH 187

REFERENCES 188

CHAPTER I

INTRODUCTION AND PROBLEM STATEMENT

Remote manipulation involves a human operator and a machine together performing a task which could be performed more easily and effeciently by the man alone, were the task or its environment not too large, small, distant, ponderous, delicate, obscure, dangerous or some combination of these. Manipulators such as the Model 8 (see Figure 1) were devised after World War II, when the Argonne National Laboratory needed ways of performing experiments with radioactive materials.[14]* The Model 8 consists of identical master and slave ends, the former grasped and moved directly by the operator. Present day manipulators are similar to the first ones in most respects, although the geometry may conform more to that of the operator's arms and shoulders, and power assist may augment his muscles. Manipulators are used in quite complex hot lab experiments,[42] for underwater retrieval, and for complete operation and maintenance of large radioactive research installations for extended periods of time,[17] to name a few examples. Their future in an increasingly technological society seems assured, for man continues to press his capabilities farther out into distant and hostile environments. Yet the capabilities of manipulators remain extremely limited, and much effort is being expended to improve them.

* Superscripts refer to references listed following the Appendices.

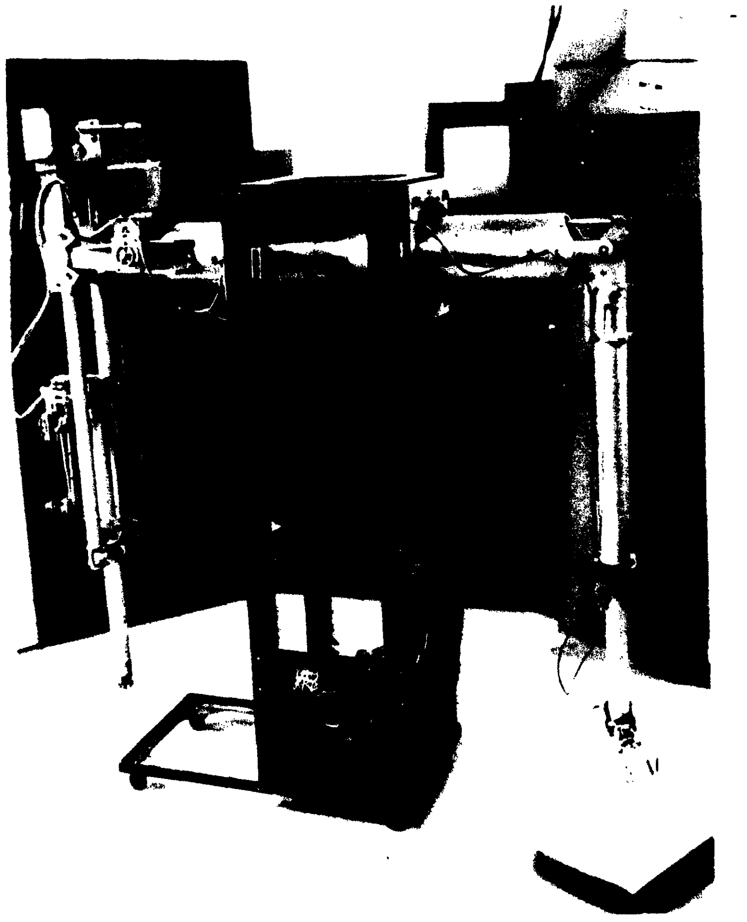


Figure 1

MASTER-SLAVE MANIPULATOR, AMF MODEL 8. THE MASTER END IS AT THE LEFT, SLAVE END AT THE RIGHT. THIS IS THE UNIT USED BY ERNST.[10]

This thesis is concerned with making a man-computer-manipulator team to perform Supervisory Controlled Remote Manipulation. The manipulator's hand _and_ the task site are considered as a system to be controlled by the operator, with the aid of the computer. This approach differs from previous work, in which only the manipulator hardware is included in the system model. The manipulator-task system is handled from the point of view of Modern Control Theory: the system is to be transformed from the current state (configuration of objects and hand) to another, desired state. An important conclusion is that the use of such methods offers a considerable improvement over manual control of the manipulator. A method akin to Dynamic Programming is used to devise motion strategies for the objects and hand. This analytic approach seems superior, in terms of computer time and likelihood of success, to similar work employing Heuristic Programming to elicit strategies.

The remainder of this chapter is a review of previous work in this field and current work in allied fields, plus a brief look at the solution method. Chapter II explains the state space solution method in detail, a group of examples following in Chapter III. Chapter IV contains a discussion of related numerical techniques. Extensions of the method and the involvement of the human operator are discussed in Chapter V. The apparatus used to "cut theoretical teeth" and demonstrate the principles is described in Chapter VI, while Chapter VII contains a brief look into the future.

Review of Previous Work

The technology of remote manipulation is currently branching in two ways. One way is typified by the work of Mosher[26][27], Bradley[5], and others striving to integrate the operator into the manipulator controls so intimately that his sense of remoteness disappears, thereby hopefully improving the limited performance presently attainable. The by-words of this work are force feedback and spatial correspondence. The master and slave portions of the manipulator, as in the original Model 8, are geometrically similar, if not identical. Regardless of the amount of force amplification provided, a portion of the required vector force is displayed directly to the operator's body at geometrically corresponding points through force-reflecting servos. A television camera mounted on the slave moves with the operator's head and gives the operator the same view he would have if he were operating the slave directly at the remote site. This seems to promise giving the operator the feeling that he is in fact at the remote site. Such a device contemplated for use in outer space has been named the Telefactor by Bradley. (Bradley, op. cit.)

Portions of such devices have been built and tested. While the head-slaved TV has yet to live up to all of its expectations (Johnsen, op. cit.), an apparatus embodying the force-reflection-kinematic similarity idea has enabled operators to perform difficult tasks in which force information is particularly important to the operator, such as spinning a hula hoop over one manipulator arm or inserting a long rod into a pipe. There is room for doubt, however, as to whether a Telefactor will work. In space applications, time delay has disruptive effects. Time delay can arise from simple transmission time or from the time needed to process telemetered data in and out of a shared channel. It was once thought that remote manipulation with time delay between master and slave would be impossible under manual control: the delay might cause the closed loop consisting of operator-master-slave (see Figure 2) to become "unstable" in some sense.

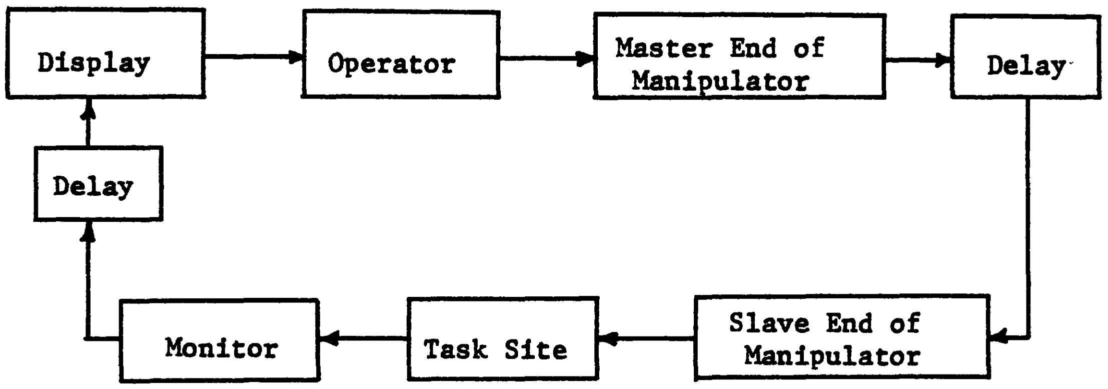


Figure 2

MANUAL CONTROL LOOP FOR MASTER-SLAVE REMOTE MANIPULATION

Ferrell[11] showed that an operator of a position-controlled manipulator having no force feedback could avoid instability and perform tasks requiring considerable accuracy simply by opening the loop, by taking his hand off the control handle from time to time while the remote end came to rest. The alternative to this move and wait strategy would be to move continuously but so slowly that the distance travelled by the remote end during one delay time were manageably small for the required accuracy. Longer delays would obviously force the user of such a strategy to move ever more slowly. Of course, task completion time under the move and wait strategy increases linearly with delay (Ferrell, Ref. 11), but the strain on the operator is small.

Now the rub is that the great benefits of force feedback accrue only if the operator keeps his hand on the control handle all the time. Ferrell has also shown[12] that remote positioning is possible with direct force feedback "in spite of delay", if the operator uses a move and wait strategy most of the time. "However," Ferrell adds, "[force feedback's] usual primary advantage, the tight closed loop control over force that it gives the operator, is lost with delay and there is the danger of unstable movements, especially those resulting from unexpected collisions". Thus a new approach is needed to improve man's ability to manipulate where delay is a factor.

The second branch in manipulation technology arose partly from the above considerations. Even without delay, however, remote manipulation is difficult. The apparatus lacks dexterity and delicate touch feedback. Vision is limited, might be intermittent in space applications or, in underwater applications, completely obscured. A skilled operator, using the most advanced force-reflecting manipulators under laboratory conditions, performs his work at "one tenth to one fourth the speed of direct manual manipulation". (Goertz, op. cit.)

Sheridan[35] proposed the Supervisory Controlled Manipulator, a device equipped with some limited intelligence of its own at the remote end, a small computer. This computer could respond quickly to emergencies with simple reflex actions or could act as an interpreter and editor of sensor data. From this start grew the concept of a man-computer-manipulator team (see Figure 3) in which the man could issue "commands" of some sort, the local computer would figure out how to accomplish them, the manipulator would act under computer control, and success or failure, with status information based on sensor data processed by the remote computer, would be returned to the operator. The saving in task completion time might be great, since the amount of transmission delay would not be a major factor. (Preliminary results obtained by McCandlish[20] do not support this, however.) A remote computer would save costly transmission of detailed operator commands and sensor feedback. The operator would be saved the strain of constant attention required under manual control. The presence of the operator, on the other hand, would allow a smaller remote computer than would a fully automatic manipulation system. The latter would either have to be preprogrammed to handle all possible contingencies or be endowed with a great deal of intelligence of its own.

The major achievement of a supervisory controlled manipulator, however, would be its very nature as a team in which each element performed the part best suited to his or its abilities. The operator, having flexibility, foresight, ability to vary his responses, and knowing what he wants done, sets significant but sufficiently simple goals for the computer-manipulator. The latter factors the stated task into a string of subtasks, each capable of direct execution.

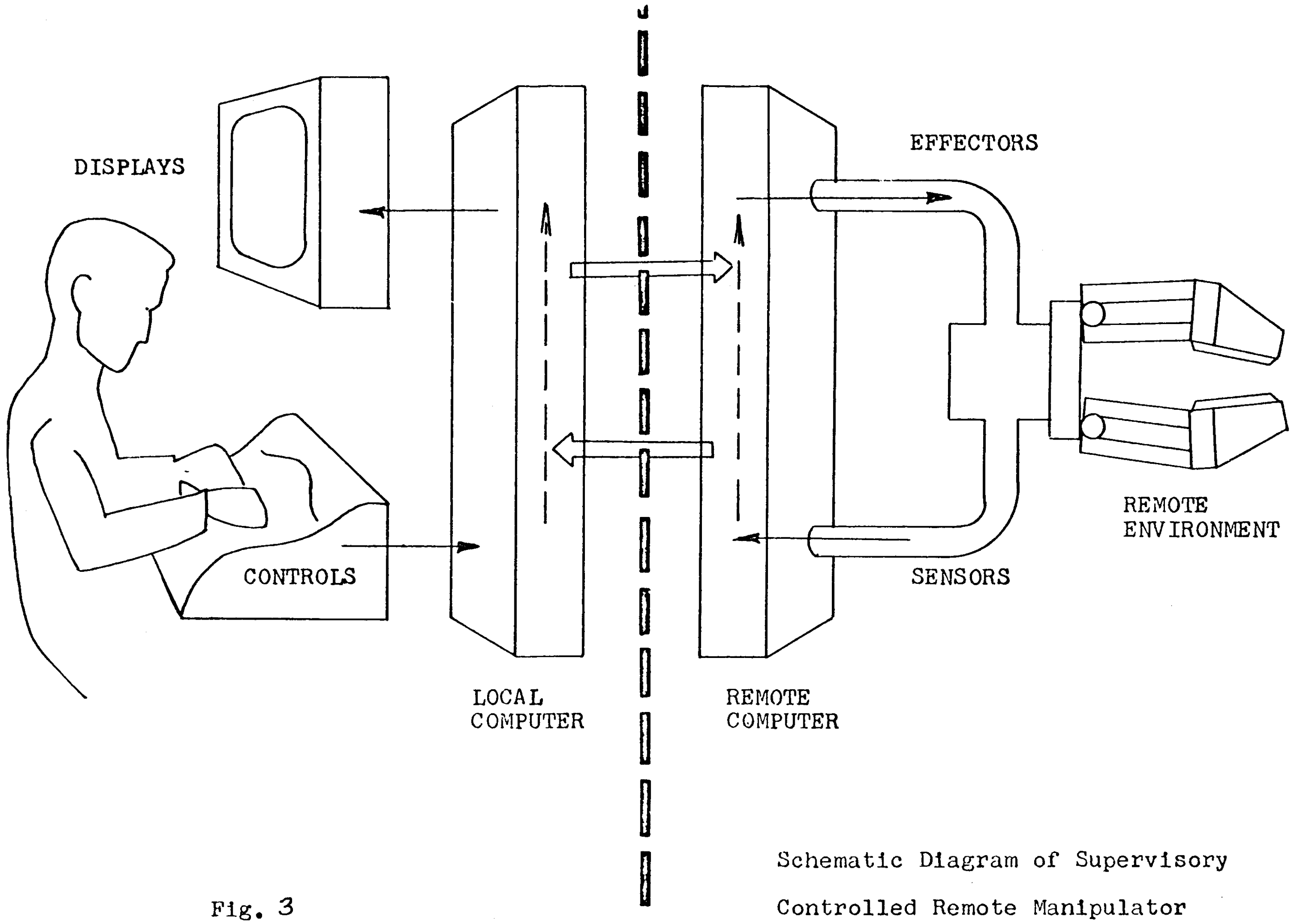


Fig. 3

Schematic Diagram of Supervisory Controlled Remote Manipulator

The plan thus formed (and approved by the operator) is set in motion under his supervision. He monitors its progress and helps in case of trouble by offering substitute goals or by taking over manually.

The Telefactor is thought of by its originators as a servo which follows the operator's motions. The manipulator itself is the system to be controlled. The supervisory controlled manipulator, together with its environment, the task site and the objects to be manipulated, can also be thought of as a system to be controlled by the operator, although the kind of control exercised is obviously different. The operator does not merely wave his arms, or attempt to describe the arm motions he wishes the manipulator to enact. His commands are likely to be verbal in nature, comprising symbolic references to names of objects, locations and so on.

The main purpose of this research has been to establish a fairly formal description of manipulation tasks so that methods of controlling such a system could be devised. The following assumptions were made:

1) The operator is equipped with a large computer and the distant manipulator is equipped with a small computer.
2) There is limited communication between the operator and the remote site.
3) The remote site constitutes a well-formed environment. That is, it is limited in extent and complexity; the objects are designed to be grasped readily by the manipulator and are to be found and moved in a more or less concave region (much like a well-designed console), except that the manipulator could be forced to reach

around an obstacle to grasp an object.

4) The operator is to tell his computer what he wants done or how he wants the environment altered. His computer, acting on data available from the remote site, designs motions and actions for the arms and jaws of the manipulator so as to accomplish the operator's desire, perhaps checking back with the operator for help or approval.

5) The operator is normally not to be involved with the detailed actions of the manipulator or with evaluation of gross feedback such as arm position, contact with objects, collisions with obstacles, and so on.

6) The remote computer must evaluate the gross feedback and take stop-gap action where necessary. The computers must decide when to ask the operator for help in recovering from disaster or to evaluate fine feedback such as texture or shape identification of unfamiliar objects.

7) The system should be capable of carrying out fairly general tasks in real time. These tasks should not be preprogrammed.

This type of system should be distinguished from preprogrammed machine tools and materials handling machines such as "Unimate."[40] While they take their instructions from an operator by manual or symbolic inputs, they are incapable of flexibility of response or communication with the operator during task execution. Note, too, the division of labor contemplated between operator and computers: the operator is the goal setter and evaluator of difficult patterns; the computers set simpler goals, do routine work and recognize simple patterns. It is felt that this is an appropriate division.

Considerable work has been and is going on in this field. The first automatic manipulation under computer control was carried out by Ernst.[10] The goal of his research was to investigate ways of equipping a computer with the ability to discover facts about its environment and use these facts to alter the environment upon command, all using hardware at its disposal. Ernst wrote an interpretive language in which he could compose programs for carrying out specific but non-trivial tasks. An AMF Model 8 manipulator, equipped with electric motors and touch sensors, was attached to the computer. The programs were designed so that the computer might be able to respond flexibly and help itself out of trouble, but no intercession by the operator was provided for. Ernst pointed out the computer's need for some internal model of the environment, although he did not describe the form of model he used.

McCandlish (op. cit.), Rarich[34] and Barber[2] have investigated various aspects of supervisory controlled manipulation at M.I.T. McCandlish simulated a rate controlled two dimensional manipulator on a computer. The operator viewed a symbolic sketch of the system on a cathode ray oscilloscope display. Extensive experiments showed that a move and wait strategy with rate control could overcome transmission delays. Supervisory controlled manipulation was simulated by providing subroutines to carry out the exacting portions of the test task. While these subroutines did not significantly reduce task completion times, they made the task so much easier, even with a delay of 12.8 seconds, that the operators relaxed and consequently made more errors than with delayed manual control! Apparently the precision required of the operator's judgements was reduced but not eliminated. Rarich composed an input language similar in some respects to Ernst's, but capable of being accepted in real time by the computer. The computer was equipped to display the status of touch sensors and report success or failure. Barber composed an input language more like FORTRAN, capable of accepting (in real time) routines with logical structure and branching conditioned on the task environment. No extensive experiments have been performed with either of these languages.

The problem of guiding a multidegree-of-freedom manipulator, which is a sub-problem of the work reported in this thesis, has been attacked as a "classical" Optimal Control problem by Mergler and Hammond.[22] They demonstrated that, even when the manipulator was redundant (so that some degrees of freedom could undergo arbitrary motions in spite of the task), a computer could (in real time, again) plan time histories for all the degrees of freedom, making the best use of the redundancies, to take the manipulator jaws from one location to another. The computational scheme involved judging competitive paths against a minimum cost criterion. The authors observed that the resulting paths were not too satisfactory and correctly blamed the cost criterion.

Tomović and his colleagues[36][37] have also pointed out the applicability of Modern Control theory to problems in prosthetics and bioengineering. (The extensions to manipulation are direct.) Tomović has built and tested an artificial prosthetic hand which has touch sensors and will grasp via reflex action an object which is touched. All of this work is confined to controlling the manipulator or prosthesis itself, without reference to a task or to a man-machine dialog.

Currently, McCarthy[21] is working on "humanoids" which take orders from human supervisors. Minsky[24] of M.I.T.'s Project MAC is building an autonomous robot, complete with vision and hopefully able to act and manipulate intelligently on its own.

Discussion of Some Aspects of the Problem

Of the many challenges which the design of a supervisory controlled manipulator presents, two which stand out are:

How to equip the system with the ability to understand what the operator wants done, (1)

and

How to enable the system to translate the operator's desire into a plan of action which is relevant to the task environment and capable of achieving the operator's goal. (2)

This thesis concentrates on these two problems. Some general remarks are appropriate at this point.

Consider the supervisory controlled manipulator as the operator's friend, a cooperative servant. If the system were merely a manually controlled device, the "commands" we could give with our hands could appropriately be called <u>manipulator primitives</u>*,

* A better picture of this idea will emerge below.

describable verbally by such phrases as "open the jaws," "move through a 60° arc," "move 4 inches left or until you touch something," commands which, by the nature of a manually controlled device, need no further interpretation. To build the kind of system discussed in the first section of this chapter, we must transcend this kind of primitive. We would like the operator to be able to give commands at some approximation to the <u>human primitive</u> level, such as "pick up the pencil," or "put it on the table, in the center," and so on. A string of dozens or perhaps hundreds of manipulator primitives might correspond to each of these relatively simple instructions. The local computer should generate such strings to save the operator having to think up, describe verbally (perish the thought!) or manually perform the manipulator primitives himself. The operator, using human primitives, is granted two advantages:

1) He can refer to actions and objects symbolically, using their names.

2) He can address himself to <u>goals</u> at something like a human level, rather than to <u>methods</u> at the manipulator's level.

(We have merely restated the two challenges from above.)

A good manipulator servant must have the following characteristics:

A) It has a symbolic representation or model of the task site. All objects, obstacles, fixed support surfaces and effectors (jaws, tools, etc.) are represented in their proper spatial relationships.

B) It can identify goals in this model. A goal may be thought of as a particular configuration of the objects, obstacles and effectors which is of concern to the operator.

C) It understands how the effectors can alter the task site as well as how these alterations are represented in the model.

D) It can receive commands which specify goals to be achieved and constraints to be obeyed. Then, using A), B), and C), it can translate the command into an expanded equivalent: "expanded" means that strings of manipulator primitives have been substituted for the human primitive in the command; "equivalent" means that these manipulator primitives, when carried out, can be expected to accomplish the stated goal. That is, the system can make a plan for carrying out the task.

E) It can execute this plan, judging its progress against the plan's expectations, keeping track of its progress by updating the model, and asking for help if trouble develops or things do not go according to the plan.

Now we can draw a more detailed diagram of this sytem. See Figure 4. The local computer is shown receiving commands, clarifying them with the operator, sending the plan to the remote computer, and receiving the remote computer's requests for aid or reports of complete or partial success. The remote computer stores the plan,

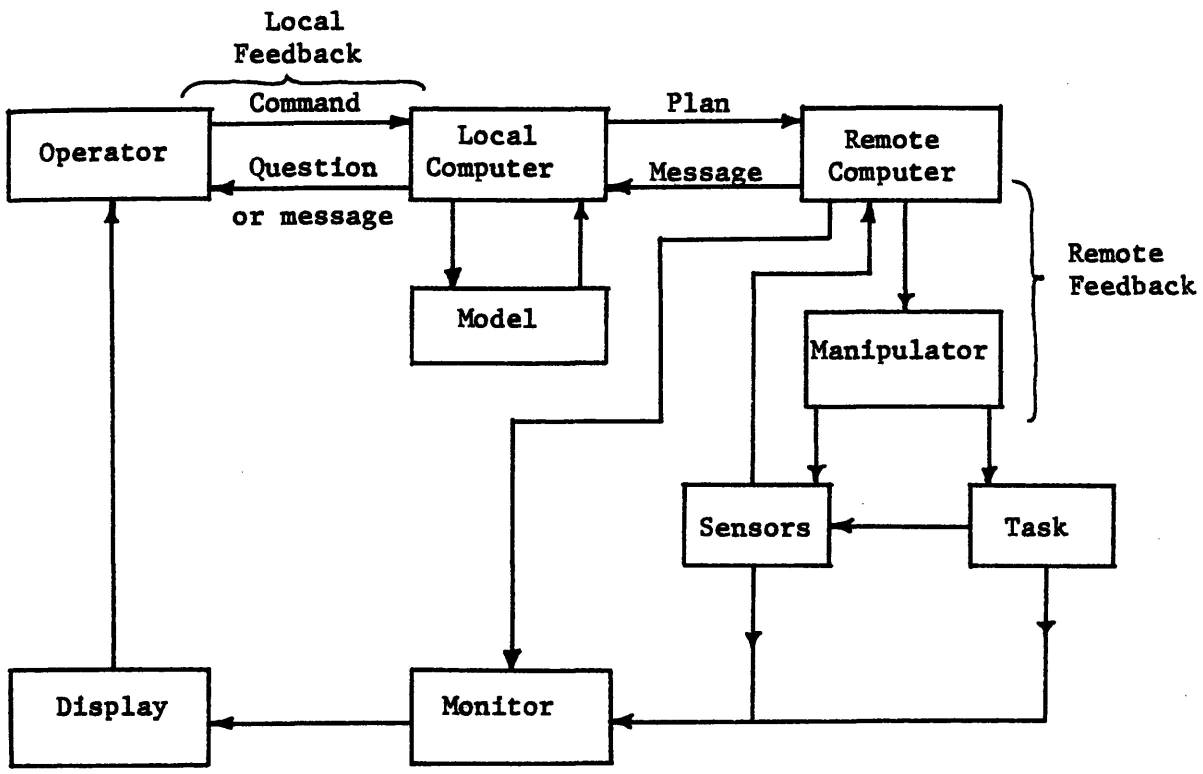


Figure 4

FUNCTIONAL DIAGRAM OF A SUPERVISORY CONTROLLED MANIPULATOR

operates the manipulator, receives sensor data, and aids in display presentation. The display closes the outer feedback loop.

Similar Work in Allied Fields

The challenges posed above bear some similarity to problems in the field of artificial intelligence.[23] Workers in this field attempt to equip a machine (usually a high speed computer) with the ability to solve fairly general problems of a limited class. Examples include the Logic Theory Machine[28] and the General Problem

Solver.[30] Mechanical problem solvers must find an efficient and efficacious sequence of elementary items (postulates, transformations and previously proved theorems; openings, moves, captures; methods of composition, decomposition, substitution, etc.) which comprises a proof, winning game, or problem solution, as the case may be. Such research usually investigates cognitive processes with the goal of ultimately producing a machine capable of autonomously solving problems as yet unsolved (although this has yet to be achieved). Occasionally the effort has been to simulate human thinking processes.

The systems created thus far are quite complex, the main difficulty being challenge (2). It is known in most of the problems studied that at least one finite* solution sequence exists. Were the sequence not too long and the alternatives at each step too numerous, direct enumeration of sequences would be a good solution method. Since an intelligent human could reject the vast majority of the proposed solutions after seeing the first few steps, efficiency and esthetics demand a better way.

One way is to test each proposed element for its ability to contribute effectively. Unfortunately, this is difficult or impossible in most artificial intelligence problems. However, because we can make a direct geometric model of a manipulation task, it is relatively easy to subject a proposed manipulator primitive to such a test. As a result, standard hill-climbing techniques are available to us. The consequences of this fact will emerge below. Hill climbing is not directly applicable to chess, for example, since it is an adversary

* A finite sequence contains a finite number of elements.

game and solutions must take account of the opponent's responses. This calls for a technique called Minimax, common in Game Theory. Fortunately, in manipulation we have no adversary!

The solution method usually employed in artificial intelligence work is called Heuristic Programming. Using certain rules of thumb (heuristics), the machine selects methods from a list and attempts a direct solution. Failing that, other methods attempt to produce relevant subproblems, which are treated in turn just like the original problem, possibly being broken down still further. Generation of subproblems is one of the hardest parts, for it may not be clear which of many possible subproblems will lead most directly, if at all, toward the solution of the main problem, and the system may not know when to abandon one chain of subproblems and try another. The result is that such systems usually work a long time, by human standards, or else cannot solve much beyond the most trivial problems.

It should be noted (Ernst, op. cit.) that manipulation rarely presents unsolved problems in any practical sense. There are some obvious constraints which are common to many manipulation tasks. For example, an object must be grasped before it can be lifted; it must be touched in a particular place before it can be pushed in a particular direction. What we want is a system which can deduce specific solutions to problems posed in a certain context (environment), where the general solutions are known, at least to the operator. Such terms do not suffice to describe a theorem-proving machine, which has no model on which to map out solutions or to test steps for their usefulness.

Why, it may be asked, is a model so vital, if the constraints are so obvious? Why not just let the manipulator poke around until it has completed the task? The answer to this is the same as the answer to "Why can't one million monkeys with one million typewriters generate the works of Shakespeare?" The trouble is that 1) it would take approximately forever, and 2) there would be no way to extract the desired result from the boundless mass of irrelevant trivia (possibly destructive arm waving or trashy monkey literature) which would be produced at the same time. The model also has the virtue of being a relatively cost-free proving ground for trial solutions, a fast time scale analog (albeit in digital form) in the tradition of Zeiboltz and Paynter.[44]

A Preview of the Method

The model we have available (and will describe below) is a true metric space: it has coordinates just like physical space and we can measure how far apart the points are. Assume, for example, that we have a single object sitting on a table. See Figure 5. We want it slid to another point on the table, avoiding the obstacle on the way. Infinitely many trajectories for the object are available, of which two are shown. Naturally, the operator wants the local computer to choose a trajectory, a direct one if possible.

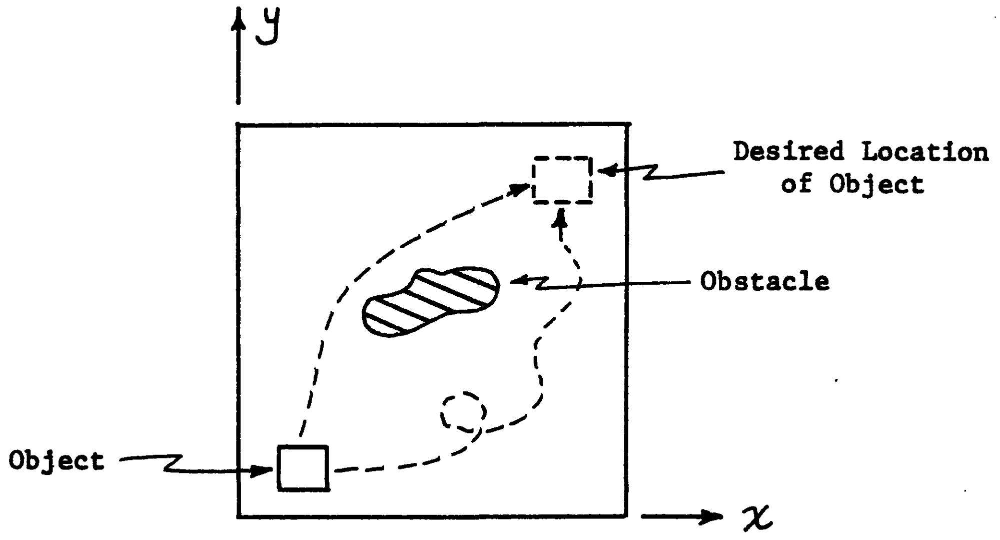


Figure 5

AN OBJECT AND OBSTACLE ON A TABLE

An engineer could identify this as a control problem: we wish to "steer" the object from one point on the table to another. If we have a force vector available with which to push the object around, we can write the equations of motion of the object in vector form as

$$\dot{\underline{X}}(t) = \underline{f}(\underline{X}(t), \underline{u}(t)) \tag{3a}$$

with

$$\underline{X} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \quad \dot{x} = dx/dt,\ \dot{y} = dy/dt, \text{ and } (x,y) = \text{the object's coordinates} \tag{3b}$$

$$\underline{u} = \begin{bmatrix} F_x \\ F_y \end{bmatrix} \quad \begin{array}{l} F_x = \text{x component of control force} \\ F_y = \text{y component of control force} \end{array} \qquad (3c)$$

and $\underline{f}(\underline{X}(t), \underline{u}(t))$ = some appropriate vector function which expresses the object's dynamics, friction, and so on. (3d)

$\underline{X}(t)$ is called the <u>state vector</u> of the system consisting of the object, because it describes where the object is plus enough dynamic information to tell us what will happen when we apply control. $\underline{u}(t)$ is called the <u>control vector</u>. The problem is then to find the appropriate control history $\underline{u}(t)$, $t_o \leq t \leq t_f$, with which to change $\underline{X}$ from

$$\underline{X}(t_o) = \begin{bmatrix} x_o \\ y_o \\ 0 \\ 0 \end{bmatrix} \text{(implying object at rest at } (x_o, y_o) \text{ at } t = t_o) \qquad (4a)$$

to

$$\underline{X}(t_f) = \begin{bmatrix} x_f \\ y_f \\ 0 \\ 0 \end{bmatrix} \text{(implying object at rest at } (x_f, y_f) \text{ at } t = t_f) \qquad (4b)$$

while constraining x and y not to take values in or too near the obstacle or beyond the edge of the table. Such problems are common, for example, in astronautical guidance and chemical process control and are solved using the theory of optimal control.[1][38] The approach is to test each possible trajectory, which fits conditions (4) and satisfies the constraints, against a cost criterion, such as

$$\text{minimize } J = \int_{t_o}^{t_f} (a\dot{x}^2 + b\dot{y}^2)\,dt \tag{5a}$$

or, more generally

$$\text{minimize } J = \int_{t_o}^{t_f} L(\underline{X}, \underline{u}, t)\,dt \tag{5b}$$

The form of the function $L(\underline{X}, \underline{u}, t)$ determines which trajectory will be selected. Since one can choose any non-negative function for L, we have considerable control over what the solution trajectory will look like.

The methods used to solve the problem include Calculus of Variations, the Maximum Principle, and Dynamic Programming. The result is the particular or "optimal" control history $\underline{u}*(t)$ which we should use. If we represent $\underline{u}*(t)$ one-dimensionally (see Figure 6), and imagine that it is made of

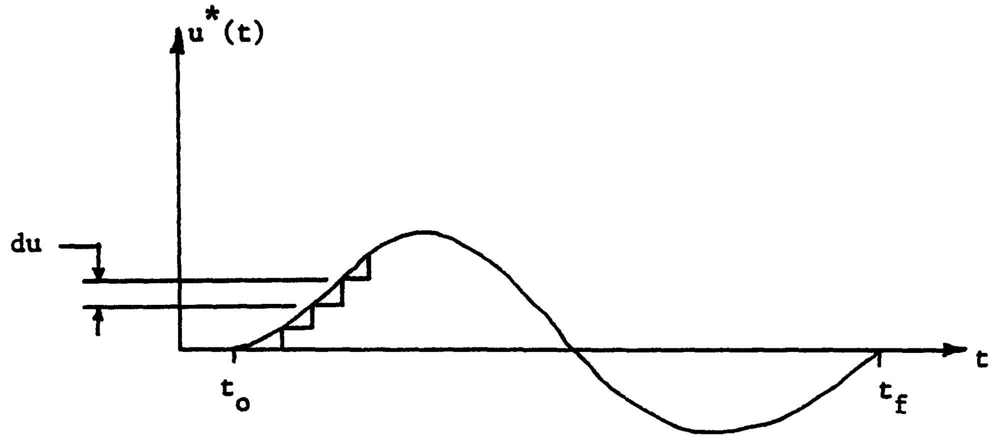


Figure 6

CONTROL HISTORY BROKEN INTO ELEMENTARY STEPS little steps of height du, then we can think of the control history u*(t) as a (continuous) sequence of elementary control actions built up by selecting, in the correct order and with replacement, from the limited set {du, -du}. In this sense, application of Optimal Control theory yields the type of elementary sequence that we discussed in connection with manipulator primitives.

To use this theory for more general manipulation problems, we must formulate our model of the system as a metric space so that we can write equations like (3) to describe the manipulator and the task site, (4) to describe what we want done, and (5) to indicate how we want it done.

The model used in this little example has nothing to do with the manipulator itself. We shall see below that the most interesting models concentrate on the objects and obstacles in the environment, and involve the manipulator only to the extent of dictating the motions of its jaws, including their grasping and releasing actions, so that task constraints are satisfied. Such constraints include 1) grasping the correct object, 2) avoiding obstacles, 3) generating "Grasp" from the correct sequence of jaw motions, opening and closing, and so on. In some environments, this will be sufficient to generate a useful solution, while in others, more details of the manipulator must be included in the model, in order, for example, that its "elbows" not strike obstacles.

The model which concentrates on the task should be distinguished from that used by Tomović or Mergler and Hammond. These workers are concerned with steering the manipulator, and use the very method indicated by equations (3) - (5) to do it. However, because the manipulator is being steered, rather than the task, and because the manipulator is thought of as a dynamic system described by differential equations, the models describing them are dynamic, concerned basically with velocities, accelerations, and forces. Optimization is usually on the basis of some convenient quantity, such as energy or time, which is relevant to such models. Optimization is sometimes used merely to absorb redundancy in the manipulator's structure, a strategy which really wastes the redundancy. The latter should either be used positively to reach into out of the way places (a task constraint), or else the manipulator should be built more simply in the first place. A dynamic model of a multidegree of freedom manipulator would in any case require a great deal of computing time and space, little of which could be directed toward the constraints of the task.

We define manipulation tasks as tasks in which the positions and orientations of objects are changed. We think of the task site as being initially at rest, and think of the result of the task as an alteration of the geometric configuration of the task site, such that it ends up at rest. This means that we specifically avoid such tasks as catching a ball or balancing a stick on end. The interesting features of task sites are therefore the static arrangement of the objects and obstacles, together with the location of the effectors which can alter this arrangement. We group these features into a set called the <u>state</u> of the system. To be sure, an object being carried in the jaws has a velocity, but we are interested in the geometric fact that the jaws coincide with part of the object and that the sequences of positions occupied by object and jaws bear the same static, geometric relation to each other all the while the object is being carried. That is, regardless of the jaws' velocity, the object is "in the jaws," and that is the task constraint that is important about carrying.

The notion of "state" appears in artificial intelligence work as well as control theory. See, for example, the General Problem Solver[30]. A problem is described by a list of features, the list comprising the problem state. The GPS attempts to reduce the difference between this state and the desired one (say, a theorem to be proved), using methods appropriate to each "difference" which can be identified. However, there is no metric for measuring such state transitions and direct analytic methods are not applicable.

The idea of a "motion space" appears in the work of Greene.[15] He was developing mathematical models of the sensorimotor behavior of infants. He modelled motion as a space consisting of "all possibilities of [motion]...without regard to the choice actually made in any one instance." Motions were to be planned by a separate "decision system" in an unspecified manner. The notion of control is not central to Greene's work, since he is primarily concerned with the existence, in a mathematical sense, of such spaces and paths in them corresponding to motion in physical space.

In the next chapter, we shall formulate our state space model of remote manipulation. Its similarities and dissimilarities to the work cited above will be clear.

CHAPTER II

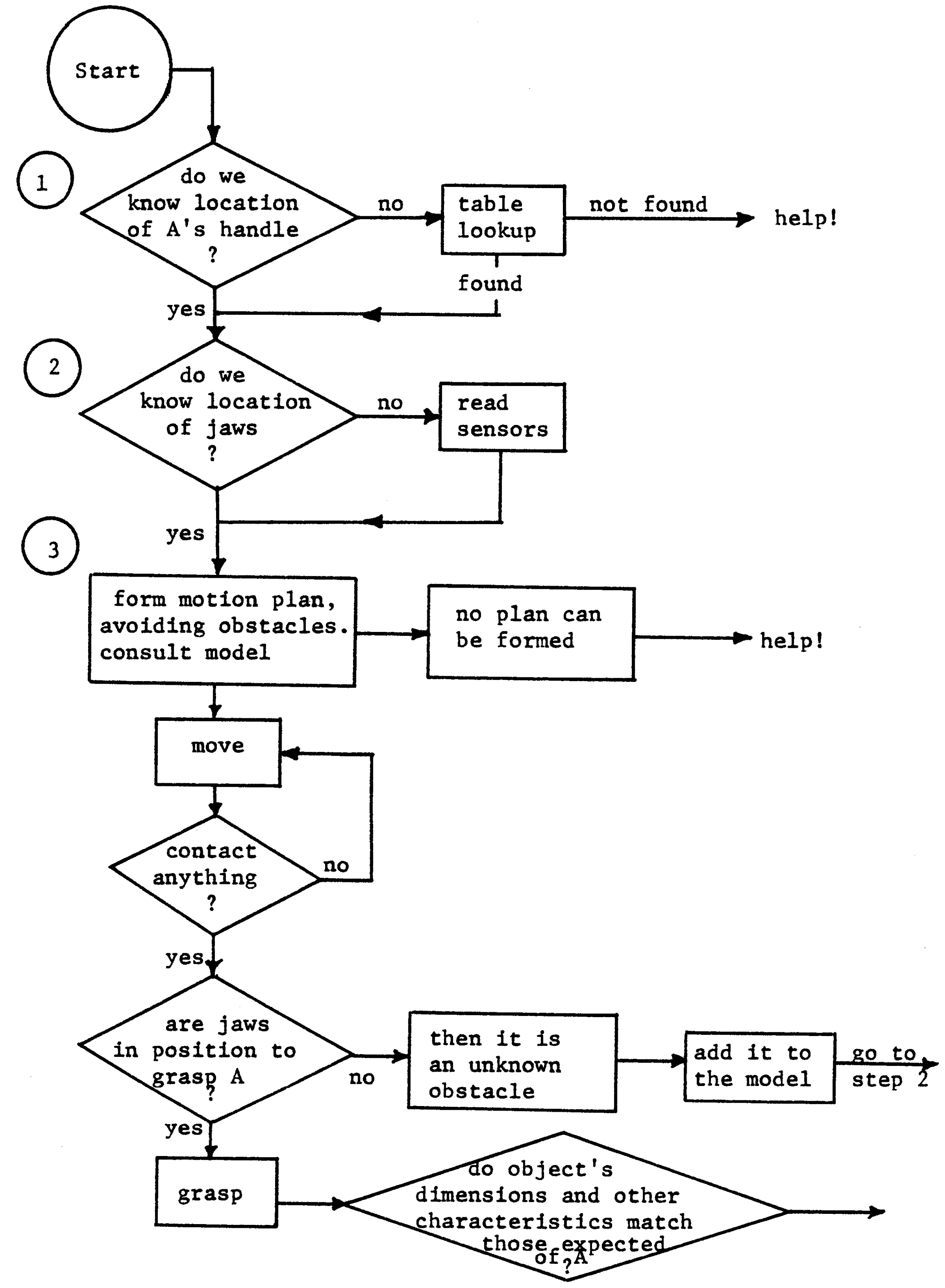
THE STATE SPACE MODEL FOR MANIPULATION TASKS

It was claimed in Chapter I that manipulation does not really present unsolved problems. The solutions, in fact, display a certain similarity, when described as sequences of such actions as: move empty jaws to X, grasp, carry to Y, release, move empty jaws to... The same few elementary motions are combined in many ways to make up complex actions, just as the few letters of the alphabet can be used to spell so many meaningful words.

The sequence of elementary motions must satisfy physical constraints, of which the foremost is that their combined result be the desired configuration of the environment. Other physical constraints include avoidance of obstacles, accurate terminal rendezvous of jaws and an object to be grasped, and so on.

On a higher (almost verbal) level, one can speak of logical constraints: the motion has to "make sense" or else the task cannot be accomplished. For example, to carry an object, the system must first know the location of the object's handle. Then the jaws must be moved there, then the handle grasped, then the object carried. At the terminal location, the system must test for support under the object, then release, then move clear. If the system tries to grasp first, then move the jaws to the object, then release, and then carry, nothing will get done. The correct sequence can easily be interpreted as a program:

CARRY OBJECT A TO LOCATION X


Start
1
do we know location of A's handle ?
no
table lookup
not found
help!
found
yes
2
do we know location of jaws ?
no
read sensors
yes
3
form motion plan, avoiding obstacles. consult model
no plan can be formed
help!
move
contact anything ?
no
yes
are jaws in position to grasp A ?
no
then it is an unknown obstacle
add it to the model
go to step 2
yes
grasp
do object's dimensions and other characteristics match those expected of A ?

... and so on. There are two key parts of this program. First is the box labelled "Form motion plan, avoiding obstacles. Consult model." Most of the action we shall study takes place here. Second is the way in which the program recovers from collisions with unknown obstacles. This might be the most difficult part of the program, except that the existence of a proper model and the box "Form motion plan..." make it easy. All obstacles are treated alike, no matter where along the path they may turn up. The planning of motion follows the same rules, regardless of whether we are making the first plan, or the one following collision with the 7th unknown obstacle. The system need know only how far along the old plan it has gone. This indicates that manipulation has some recursive properties; more interesting ones will be discussed in later chapters.

Planning and Quantization

The planning of motion can actually be quite extensive. The entire sequence of motions, including all grasps and releases, can be generated at once as part of one plan. When we apply the plan to the model, we can tell if the plan can be expected to do what we want. It is our contention that planning is essentially different from execution, although they go hand in hand. In planning, the computer interacts with the operator and an idealized version of the task site, the model. In execution, the computer must operate the manipulator and interact with the task site itself. The plan is a sort of verbal statement of how to do the task, less detailed as the level of the model's abstraction increases: "If we move like so and grab A, then move like that, shifting it to X, that should do it." During execution, these grabs and shifts could prove difficult to achieve. No amount of advance planning can guarantee success on the first try. The locations and orientations of objects cannot be known precisely, so "grasp" may fail. Vibration or collision may shake the object loose from the jaws. Barring infinite planning intelligence, the burden of handling such events must fall on the execution function. We therefore must consider planning as a model of execution.

Let us then, for planning purposes, conceptualize manipulator motions as static atoms to be strung together in an appropriate way so as to span the task which the operator specifies in the model. We should limit ourselves to as few different kinds of such atomic commands as possible. For example,

$$\begin{array}{l} \text{Open jaws} \\ \text{Close jaws} \\ \text{Move jaws one inch} \left\{ \begin{array}{l} \text{left} \\ \text{right} \\ \text{forward} \\ \text{backward} \end{array} \right. \end{array} \qquad (1)$$

Note two things about this set: First, it is static and geometric, rather than dynamic. We are interested in the static result of each action in the plan. Only during execution do we watch while each action is being accomplished, so that we may monitor progress and recover from a breakdown in the plan.

Second, the set is quantized. All points in the task site reachable by any combination of these commands lie on a grid of one inch squares. The size of one inch is illustrative only, but it seems inevitable for the models and solution methods we use that quantization at some level be employed.

Quantization in the elementary motions brings quantization to the plan, hence to the task model. If the quantization size is too large, important features of the environment or requirements of the task may fall between the points and be ignored or unexpressable. If the quantization size is too small, much computing time and storage space will be wasted, since the description will be unnecessarily detailed. The quantization need not be the same size all over: it may be small near objects or places of interest and be large in wide open spaces where there is nothing of interest.

Quantization affects the way the plan is formed, and how it is carried out. If the task site and the objects are quantized to extreme fineness, then the required jaw motions can be planned with equal fineness and, except for bad information, the plan can practically be run open loop, with little attention to feedback from the environment. But this much quantization overloads the computer. If there is no information at all concerning object location and shape (equivalently, no quantization points), there is a minimum of planning and a maximum of fumbling about. This fumbling must be organized very carefully into well-planned exploration, as Ernst did (Ernst, op. cit.), but not so well-planned that general tasks cannot be easily input and executed, or so loose that damage is done or too much time is required for execution.

In between, we have a practical quantization level, bearable by the computer, in which a desired object appears on a minimum of one quantization point. (Obstacles, <u>un</u>desired objects, may be conveniently made to "disappear" with no loss of vital information if the grid size is made large in their vicinity and the points fall around but not in them.) The required motions must now be planned using limited precision, limited knowledge of objects' size, shape, location and orientation. As a result, there may be some collisions. The jaws may not be properly aligned or located for grasping. These deficiencies in the <u>plan</u> must be made up for by more sophisticated execution, although less outright fumbling should be needed. We trade the storage and computation required by fine quantization for a less certain plan. This plan requires in turn more computation for its execution, but there is computing time available during execution, even time enough to make a new plan.

Then grasping an object may be accomplished by bringing the jaws to the best location the plan can generate; at this point, some well-planned, even rigidly patterned, fumbling commences in a limited region. The jaws are opened extra wide to allow for uncertainty in the object's size, location and orientation. The computer moves the jaws and watches the touch sensors for clues as to how the operation is progressing. This introduces yet another form of quantization, since continuous touch sensors are not available, even to people. If there is a single sensor on the inner face of each manipulator jaw, then the following four grasping situations will "look" the same, since both sensors will report contact:

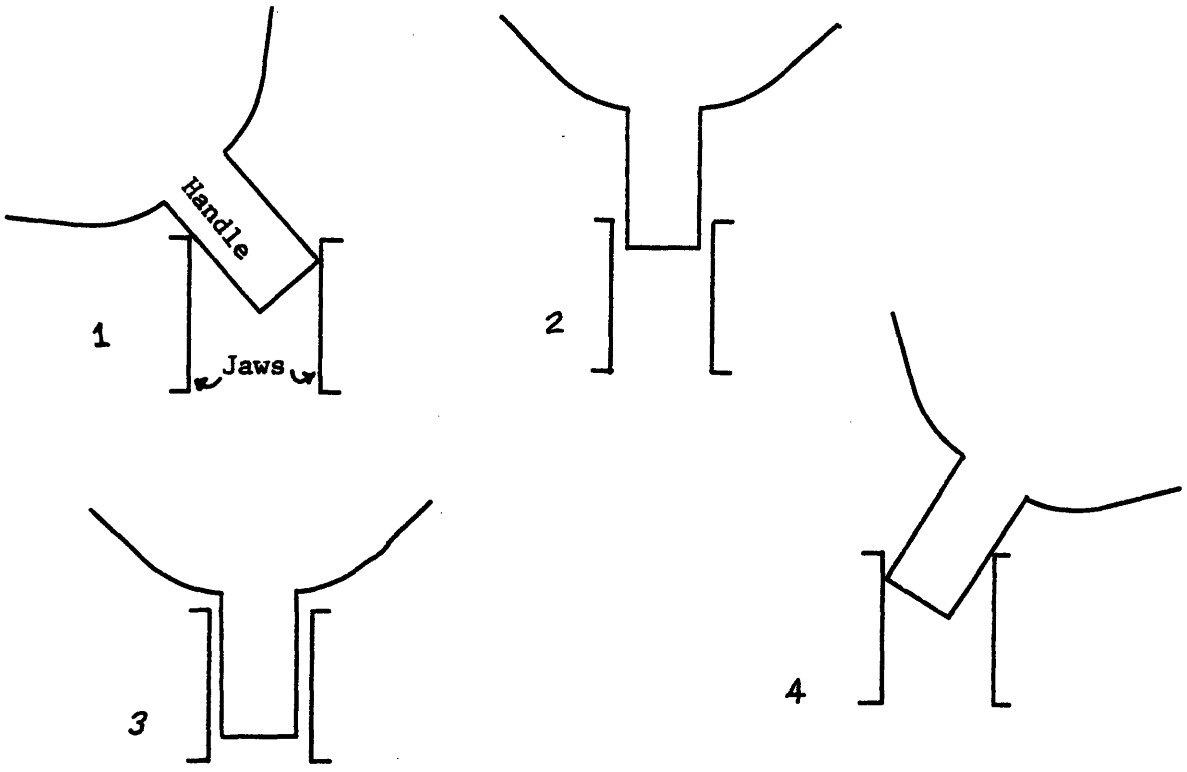


Figure 1

FOUR GRASPING SITUATIONS WHICH GIVE THE SAME TOUCH SENSOR REPORT

Of these, only number 3 is satisfactory for grasping. It appears that sensors arrayed as in Figure 2 are more likely to give meaningful grasping information. More sensor points will again put strains on the computer:

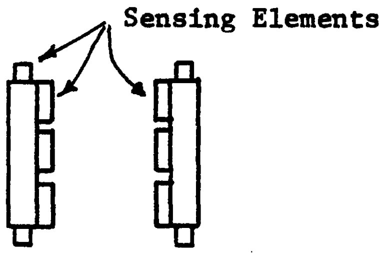


Figure 2

PROBABLE MINIMAL TOUCH SENSOR DESIGN

Note, however, that infinitely fine touch sensor quantization is not really needed, rather only enough to do the tasks we are capable of, using the elementary atomic commands at our disposal. The sensor points must be close enough together so that objects do not fall between. Several sensor points should fall on the object when both jaws and object lie on quantization points. Conversely, given workable touch sensor arrangement, the plan of gross motions can be somewhat relaxed in precision, because collisions can be sensed before damage is done, and errors in jaw position can be corrected during grasping. Thus plan, model quantization, task execution, and sensor quantization all interact: extra investment in sensors and execution strategies reduces greatly the planning effort required to manipulate.

Not all the information about an object need be described by quantization points on the model. It is easier, for example, to store in a separate table such information as the current best values of the orientation of the object's handle, the size and shape thereof, its distance from the object's center, and so on, storing on the model grid only the rough location of the object's center with no reference to handles. The plan is formed using this rough location plus the orientation and distance data about the handle, taken from the table. During execution, reference is made to size and shape information only when grasping begins, first to ensure that the jaws open wide enough, second to confirm that the correct object has been grasped. The interplay of plan and execution is a very complex one. Only the most basic tradeoffs have been discussed here. Ernst's work concentrated on execution strategies. In this thesis, the emphasis will be on planning.

The planning function must be adept at putting together strings of atomic commands. This forms the core of many artificial intelligence problems. Here, however, we have the advantage of a model on which we can _measure_ the effect of performing a given elementary motion. Our situation is thus very similar to that which arises in Optimal Control Theory, in which command elements are strung together to accomplish a control task.

States and State Spaces

Optimal Control Theory is closely linked with the concept of _state_. The state of a system is a list, called the _state vector_, of quantities (_state variables_) sufficient to tell us what we want to know about the system's configuration plus which parts of that configuration will change if we apply control. Control here means a string of elementary motions (_commands_), drawn from a set like (1). It appears that we can be somewhat arbitrary about what quantities we put into the state vector. Since elementary commands make noticeable changes in the task site, the state vector describing some task had better include the quantities relevant to that task which are subject to alteration by the allowed commands.

Since the state vector is a list of numbers changeable by the commands, we can think of the set of _all_ allowed values of the state vector as a discrete array of points, usually called the _state space_. (It is discrete because the commands are quantized.) Consider an example on the minds of many people this year in Boston, baseball. Let the system be the batter during one time at bat, and let us be interested solely in the ball and strike count. The pitcher can pitch a ball or a strike, up to 4 of the former and 3 of the latter. For elementary commands, we then have

$$\left.\begin{array}{l}\text{pitch a ball}\\ \text{pitch a strike}\end{array}\right\} \tag{2}$$

For the state vector, we have

$$\begin{bmatrix}\text{number of balls}\\ \text{number of strikes}\end{bmatrix} \tag{3}$$

For the state space, we have Figure 3.

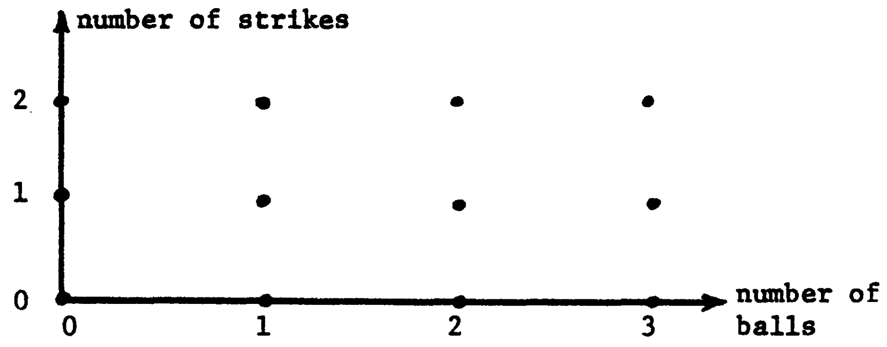


Figure 3

STATE SPACE CORRESPONDING TO BALL-STRIKE COUNT

Thus, before each pitch, including the first, the system occupies one of the states in the space, the first being (0,0). We can show the possible results of each pitch, excluding hits and other complications, by connecting certain of the states with lines, as in Figure 4. Each line implies that execution of one of the allowed commands will transform the system from the state at one end of the line to the state at the other end.

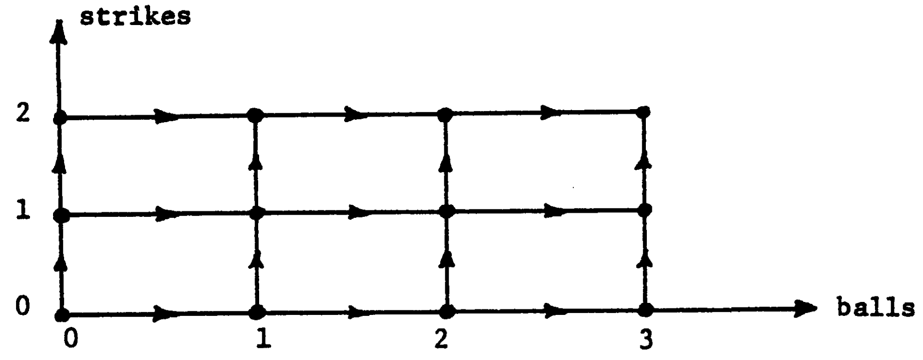


Figure 4

STATE SPACE SHOWING ALLOWED TRANSITIONS

The arrows indicate that the ball-strike count, consistently with the allowed commands, can increase but not decrease. The absence of diagonal lines indicates that, on any one pitch, the number of balls <u>or</u> the number of strikes can increase, but not both.

If we wanted to model an entire half inning of play, we would need to add at least one more state variable, the number of outs. This would require a third axis, normal to the other two, bearing values 0, 1, 2. Of course, we could expand the state vector still further if we wished, to indicate which bases were occupied, or what inning it was, or what the score was, or many others. A larger discrete space would result, again with some of the points connected by lines. It is up to us, depending on our interests, to construct the state vector as we wish. Then we imagine that the allowed commands are applied one at a time, again and again, in all combinations. This generates the state space.

In manipulation, we are interested in the positions and orientations of objects and jaws (but not their velocities or accelerations), since these are the stuff of manipulation. If we consider all possible objects at once, the space which keeps track of all their positions will be large indeed. So let us think first of one object only, and consider only its position on a table. A point in the space corresponds to the fact that the object is at a certain point on the table. Consider next one object and the jaws. A point in the space then must correspond to the fact that the object is at one point on the table _and_ the jaws are at another (but not necessarily different) point on the table. To change the state of the system from one of these points to another, we apply commands from the allowed set. The totality of points then represents all possible combinations of (quantized) object location and (quantized) jaw location which can be physically realized using sequences of the allowed commands. Thus points in this space correspond to situations which have _meaning_ in terms of manipulation, and each point represents a unique situation.

Of course, some sequences of commands make significant changes in state, changes we call _tasks_. While "open jaws" may not be significant by itself, a sequence which results in the object being shifted from one point on the table to another can be dignified with the name task, since it corresponds to a significant, if simple, human primitive.

The operator confronts the task site, or some representational display of it, while the local computer confronts the state space corresponding to relevant features of the task site. The operator wants a particular task accomplished. If the desired configuration of the task site is represented by a point in the state space, then it is easy to make the local computer understand that the operator's desire will be achieved if the system state is driven from its present location in state space to the desired one. An accomplishable task corresponds to a change in state which must span many intervening states in the space. A path or sequence of states may then be said to exist between the current state and the desired state. Each leg of the path, connecting two adjacent states, is accomplished by executing one of the allowed commands. The path reads like an ordered work description to the manipulator. It is easily coded as a short sequence of numbers and sent to the remote computer. This path is found via search of the alternatives, guided by some cost or optimality criterion. More on this below.

The elementary commands are to be accomplished one at a time, in path order, by preprogrammed, but not rigid, routines. Such routines must be capable of testing for proper completion of the command or for unexpected sense inputs. The work of Ernst shows that this can be done.

To summarize, the model is a set of all possible configurations we are interested in. Each configuration differs from its immediate neighbors in the state space model by exactly what one command in the elementary set can accomplish. Thus we may say that the model, the state space, is the set of all task possibilities achievable by arbitrary sequences of the allowed commands. The operator, by indicating a new state he wishes the task site to occupy, designates a task he wants done. He is thus in <u>control</u> of the task site, and this is what we wanted back in Chapter I.

<u>Representation of the State Space as a Finite Graph</u>

In this section, we make a formal statement of the State Space model in terms of Graph Theory.[5][32] A <u>graph</u>, G, denoted by

$$G = (X,\Gamma)$$

is a description of the relationships which a function Γ imposes on the elements x of the set X. Usually we draw the graph as a picture, with <u>vertices</u> or <u>nodes</u> representing the x's, and the relationships in Γ represented by directed line segments or <u>arcs</u> connecting some of the nodes. A graph is finite if it contains a finite number of nodes. There is an arc directed from x to y if y is an element of the set $\Gamma(x)$, which is the set of all nodes which can be reached from x in one jump. y is then said to be <u>adjacent</u> to x. See Figure 5.

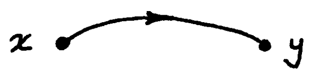


Figure 5

ARC FROM x TO y

If y is an element of $\Gamma(x)$ and x is an element of $\Gamma(y)$, then there is an arc from x to y and another arc from y to x. See Figure 6.

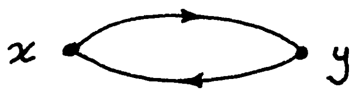


Figure 6

ARCS FROM x TO y AND y TO x

This is often condensed to a single undirected edge, as in Figure 7, although we do not make this condensation if for some reason we wish to distinguish one arc from the other.

Figure 7

EDGE BETWEEN x AND y

A sequence of arcs $U = \{u_1, u_2, \ldots\}$, such that the terminal node of u_i is the initial node of u_{i+1}, is called a path. An arc from node x to itself is called a loop, and a path from x which eventually returns to x is a cycle.

A directed graph contains only arcs, while an undirected graph contains only edges. A mixed graph may contain some of both. (For example, a city street map in which some streets are one way may be represented by a mixed graph in which intersections of streets are the nodes and streets are the edges or arcs.)

Graphs are used to represent chemical compounds, computer programs, manufacturing processes, puzzles and games, etc. Graphs are appropriate in problems in which connectivity, relatedness, adjacency, distance, combinations, or like concepts are of interest. This makes graphs ideally suited to represent manipulation task situations.

Directed graphs are useful to describe problems in order or dominance relations like:

> The Arab ambassador and the Israeli Ambassador don't like each other
>
> The Russian ambassador is senior to the Canadian ambassador
>
> The Slabovian ambassador's wife is in love with the Transylvanian ambassador
>
> .
>
> .
>
> .

Can the Chief of Protocol seat all the ambassadors and their wives at one table without insulting or embarrasing anyone? Such a problem can readily be solved if there are no cycles in the corresponding graph[7], but the algorithms break down if there are cycles or if the graph is mixed.

Undirected graphs arise in maze problems, for example, where the vertices are corridor junctions and the edges are the corridors. Then we may ask for a path which leads to the exit. Algorithms exist for finding such a path (Berge, op. cit.). The kind of problem we will deal with is one in which there usually exist many paths between two vertices. Then we may ask two related questions:

1) How do we eliminate redundant paths between two points?

2) How do we find a path most to our liking?

The answer to 2) contains the answer to 1).*

Let us start with about the simplest physical manipulation space, a line on a table. On this line lies an object. The manipulator jaws can move along the line, open and close. See Figure 8.

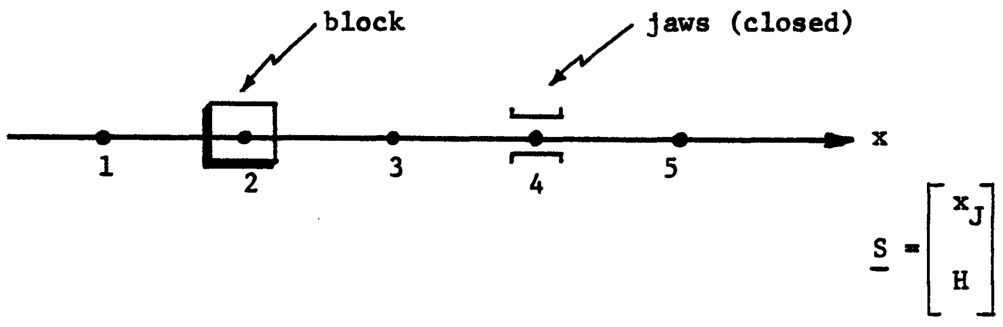


Figure 8

PHYSICAL SPACE

Thus we are equipped to manipulate the object from one of the five designated points on the line to another. The graph or state space we are about to draw will contain some of the logical and physical constraints required to accomplish all the manipulation tasks

* Appendix I states the results of this chapter in more formal mathematical terms.

possible in this limited context. Let us, as a first approximation, take for state variables the location of the jaws and an indicator which tells whether the jaws are open or closed:

$$\underline{S} = \begin{bmatrix} x_J \\ H \end{bmatrix} = \text{state vector}$$

where

$$x_J = x \text{ coordinate of the jaws, } x_J = 1, 2, \ldots, 5.$$

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

It is probably true that no simpler state vector exists which will allow even a semblance of manipulation to be planned in this physical space. If one omits H, one can only steer the jaws around, (cf. Mergler and Hammond or Tomović and Petrović, op. cit.) but cannot express the notion of grasping, which is fundamental to manipulation. If one substitutes the object's coordinate for that of the jaws, one can plan motions of the object once it has been grasped, but the logical problem of expressing the sequence "move empty, grasp, carry..." is not solved. We shall develop a solution step by step in the next few pages.

The elementary commands which are relevant in this context are those which make unit changes in the elements of the state vector. Thus:

$$\text{Elementary commands} = \begin{cases} \text{Open jaws} \\ \text{Close jaws} \\ \text{Move jaws one unit right} \\ \text{Move jaws one unit left} \end{cases}$$

Applying these commands is the only way to alter the state variables, hence the only way to make changes in the physical space. Since we allow 5 values for x_J and two for H, the graph or state space (Figure 9) has 10 states:

$$\text{Allowed Commands} = \begin{cases} \text{Open} \\ \text{Close} \\ \text{Move right 1 unit} \\ \text{Move left 1 unit} \end{cases}$$

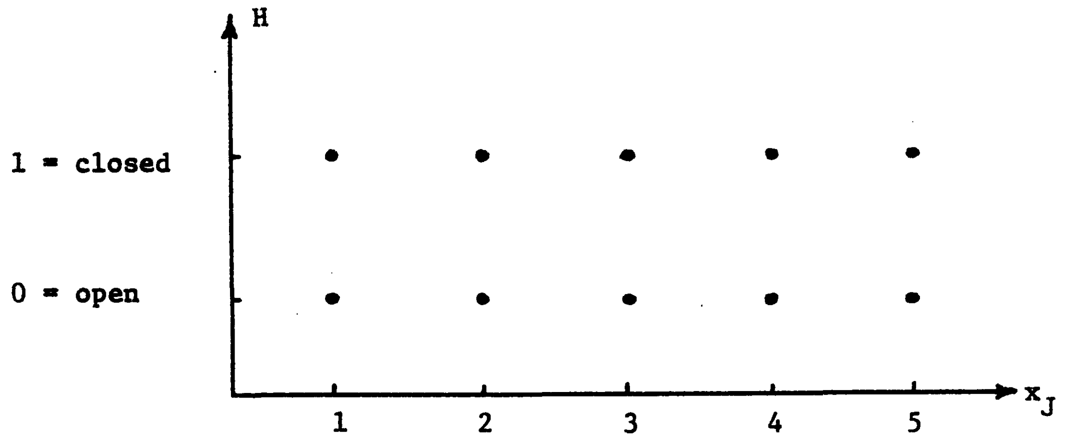


Figure 9

STATE SPACE CORRESPONDING TO FIGURE 8

Note that the state space has more dimensions than the corresponding physical space. This is typical of such spaces and will cause us some grief later on.

By inspecting the set of allowed commands and the environment, we can deduce what commands can be executed at each point in state space, and which cannot. These crucial distinctions can be made at each state without reference to what is possible at any other state. Thus we obtain Figure 10:

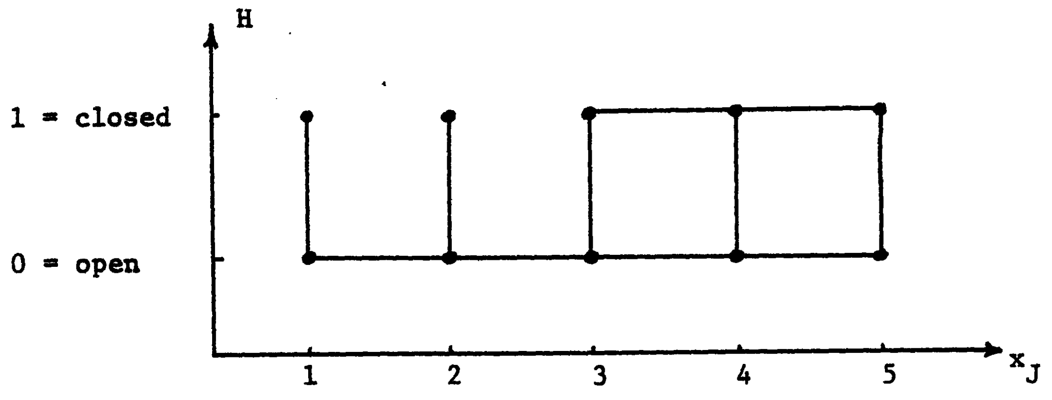


Figure 10

ALLOWED TRANSITIONS OF $\underline{S}$ BASED ON FIGURE 8

The existence of a horizontal edge between two states implies that the jaws may move in physical space between the corresponding locations. A vertical line means that the jaws may open or close at the corresponding point in physical space. The two missing lines show that the closed jaws, when in locations 1 or 3, cannot move to location 2, because a collision will occur with the object. If the object were unknown, these two lines would be present. The jaws, equipped with touch sensors, would discover the object in time and its presence would be denoted in the state space by the deletion of these two lines. The system, according to Figure 8, currently occupies state $\underline{S} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$. Say we want the jaws to grasp the object. This means we want the system to occupy state $\underline{S} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$. Thus we have demonstrated, for this simple example, the ability of the state space model to represent a task statement and to embody the physical constraint of obstacle avoidance.

The Task Plan Found Via a Shortest Path Problem

Now, how does the local computer figure out that the jaws must move over, open, and straddle the object, then close, this being the obvious logical requirement for accomplishing the operator's desire? The procedure is to assign some length or cost to each allowed transition on the graph, basing these costs, not necessarily on any physical concept of distance, but rather on how in general we would like the task carried out, still without dictating the details of the solution. For example, opening and closing are cheap in fuel and not too dangerous to successful completion of the task, so each open-close edge is priced the lowest, one unit. For esthetic reasons, we deem it inappropriate for the jaws to move about wide open, except when necessary, so we charge less for motions of the closed jaws (horizontal lines for which H = 1) than we do for motions of the open jaws. This is inconsistent, as the careful reader has probably noticed, with the reasonable notion of charging more for carrying the object in the closed jaws than for motions of the empty open jaws alone. This will be remedied shortly. In Figure 11 we show this structure of costs:

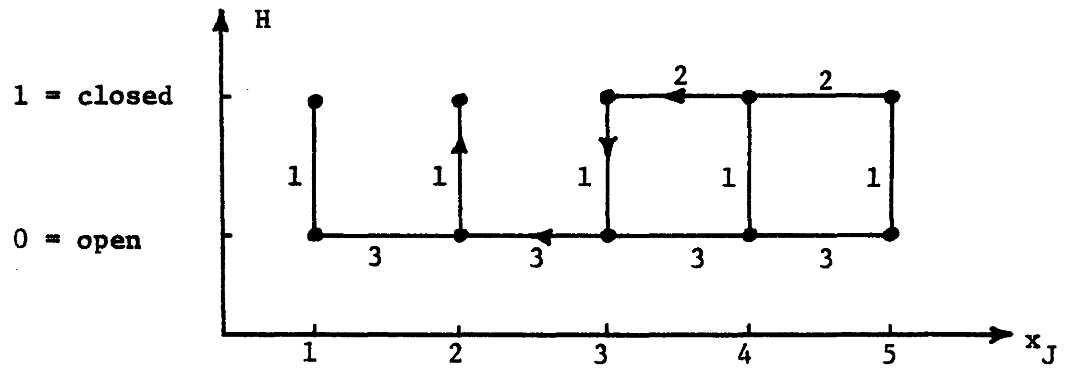


Figure 11

STATE SPACE WITH COST STRUCTURE AND A PATH FROM $\underline{S} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$ TO $\underline{S} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$.

Costs may be assigned for a wide variety of reasons, some of which we indicated above: risk, energy or fuel, time, distance, or even esthetics. These costs may be assigned uniformly to each edge on the graph representing execution of a particular elementary command, as we do in Figure 11, or we may charge more in some regions of state space than in others if there is a good reason for doing so, such as increased risk, or insufficient information concerning the physical environment in the corresponding physical areas. The cost values may be arbitrary, or may be derived from physical considerations, or may indeed represent the desires, whims or even fears of the operator.

As the reader must by now suspect, we then ask the computer to find the shortest (cheapest, safest, fastest, prettiest) path in state space from the current state ($\underline{S} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$) to the desired state ($\underline{S} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$). The resulting path will do its best to avoid the costly regions or commands as much as possible. By assigning the costs, we thus have considerable control over the characteristics of the winning path. In Figure 11 above, the shortest path is indicated by arrows. Reading the path in order, we obtain the following work description for accomplishing the original task of grasping the object:

Move jaws one unit left

Open jaws

Move jaws one unit left

Close jaws

Done.

This path demonstrates that we can express in a graph state space the logical constraints involved in moving the jaws to an object and grasping it. This is the first of many interesting logical situations we can model this way.

Another logical problem is solved automatically when we ask that the jaws move from location 4, closed, to location 1, closed. This translates to: change $\underline{S}$ from $\begin{bmatrix}4\\1\end{bmatrix}$ to $\begin{bmatrix}1\\1\end{bmatrix}$. The shortest path is shown in Figure 12, and renders the work description

Move left one unit

Open

Move left one unit

Move left one unit

Close

Done.

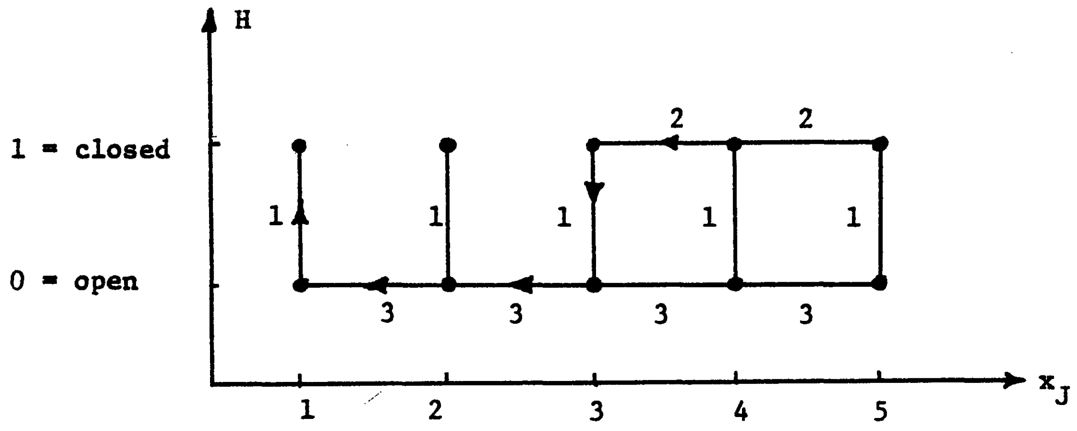


Figure 12

A SHORTEST PATH FROM $\underline{S} = \begin{bmatrix}4\\1\end{bmatrix}$ TO $\underline{S} = \begin{bmatrix}1\\1\end{bmatrix}$

The system "understands" that to move the jaws past the object, it must open the jaws to straddle the object. Note that this same group of commands, executed in another order, might be so irrelevant to the desired task as to push the object off the table. Note, too, that the more expensive but equally efficacious alternative path, consisting of opening first and then moving three units left, was avoided, along with all problems connected with bumbling about, moving in wasteful circles, and other ineffective motions. Last, if we code the commands in the allowed set using two bits:

open = 00

close = 01

move right = 10

move left = 11

then the path we just found can be represented compactly and unambiguously by

11, 00, 11, 11, 01

This brief sequence of bits may be telemetered very cheaply, in terms of power and bandwidth, to the remote computer for execution.

It is important to notice, throughout all of this, that optimality, _per se_, is not our foremost goal. Rather, our goal is to find a fairly direct path with some desirable characteristics. Optimality criteria are our tools for accomplishing this. Our problem is not so much to create a path from nothing as to cull a reasonable path from countless millions of competing alternatives, most of which do nothing. Optimality criteria are admirably suited to doing this.

The graph we have been using suffices to get the jaws to the object, but as it stands, there is no way to express carrying. This is because we have not distinguished motions of the empty jaws from motions of the jaws when grasping the object. To remedy this, consider first how the graph in Figure 11 would look if the object were in location 1 rather than in location 2. See Figure 13:

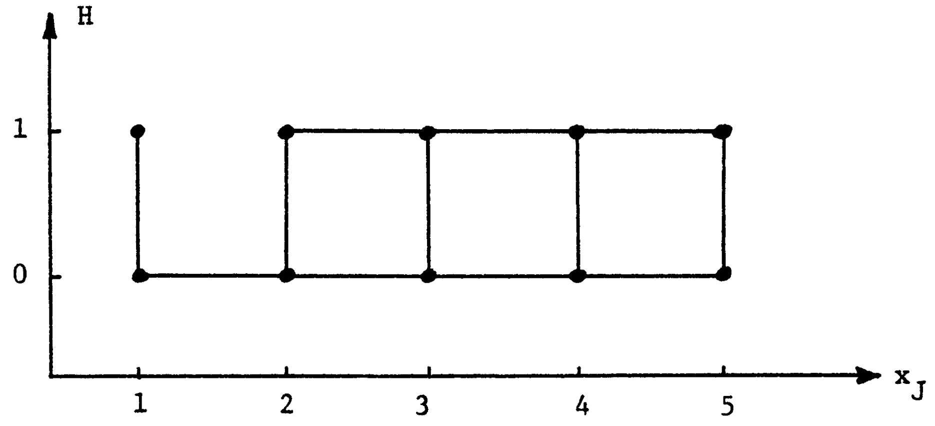


Figure 13

STATE SPACE WITH OBJECT IN LOCATION 1

Imagine now that the state vector takes the value $\begin{bmatrix}2\\1\end{bmatrix}$ in Figure 11, indicating that the jaws have grasped the object. Imagine further that the object is then carried to location 1, but that we represent the result of this by giving the state vector the value $\begin{bmatrix}1\\1\end{bmatrix}$ in Figure <u>13</u>. If we draw Figures 11 and 13 together, with an edge labelled "Carry" joining $\begin{bmatrix}2\\1\end{bmatrix}$ in Figure 11 and $\begin{bmatrix}1\\1\end{bmatrix}$ in Figure 13, then we get Figure 14, which represents pictorially what we have just said verbally:

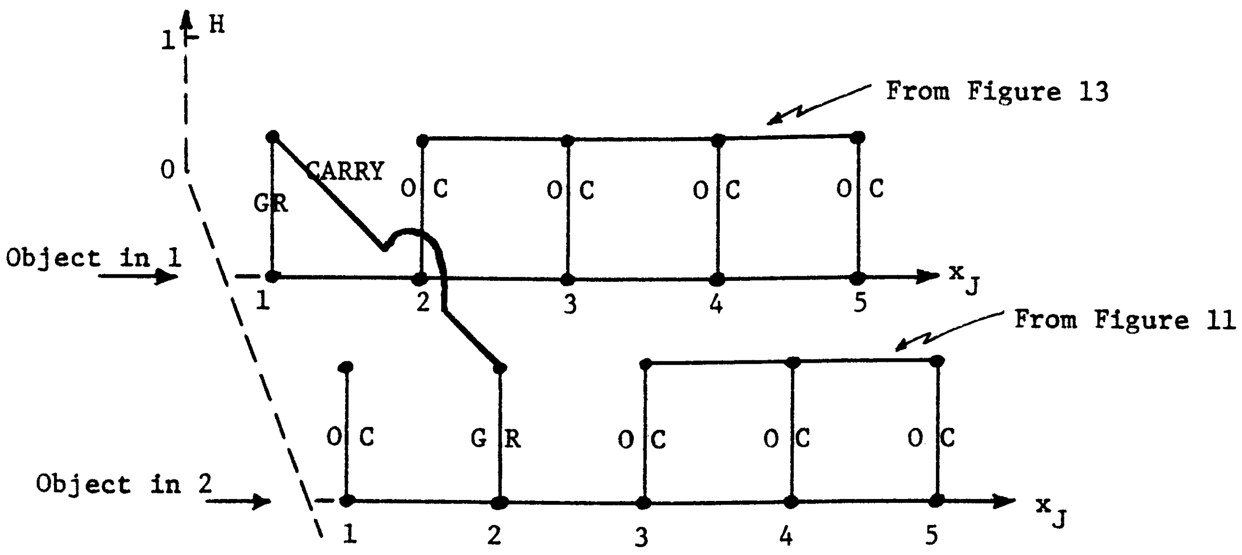


Figure 14

TWO STATE SPACES JOINED TOGETHER

The edges labelled "GR" correspond to the commands "Grasp" and "Release", while the edges labelled "OC" correspond to "Open" and "Close". It is easy to generalize from Figure 14 if we recognize that its two parts each correspond to different values of a new state variable, the object's location, denoted by y_o. The state vector then becomes

$$\underline{S} = \begin{bmatrix} x_J \\ y_o \\ H \end{bmatrix} \quad \begin{array}{l} x_J = \text{x coordinate of jaws, } = 1, \ldots, 5 \\ y_o = \text{x coordinate of object, } = 1, \ldots, 5 \\ H = \text{status of jaws, } = 0, 1 \end{array}$$

We also have the new commands "Grasp", "Release", "Carry object left one unit", and "Carry object right one unit". "Carry" is, appropriately, the only command which can change y_o. Naturally, it changes x_J simultaneously. When we draw all 5 possible versions of Figure 11 together, we get Figure 15:

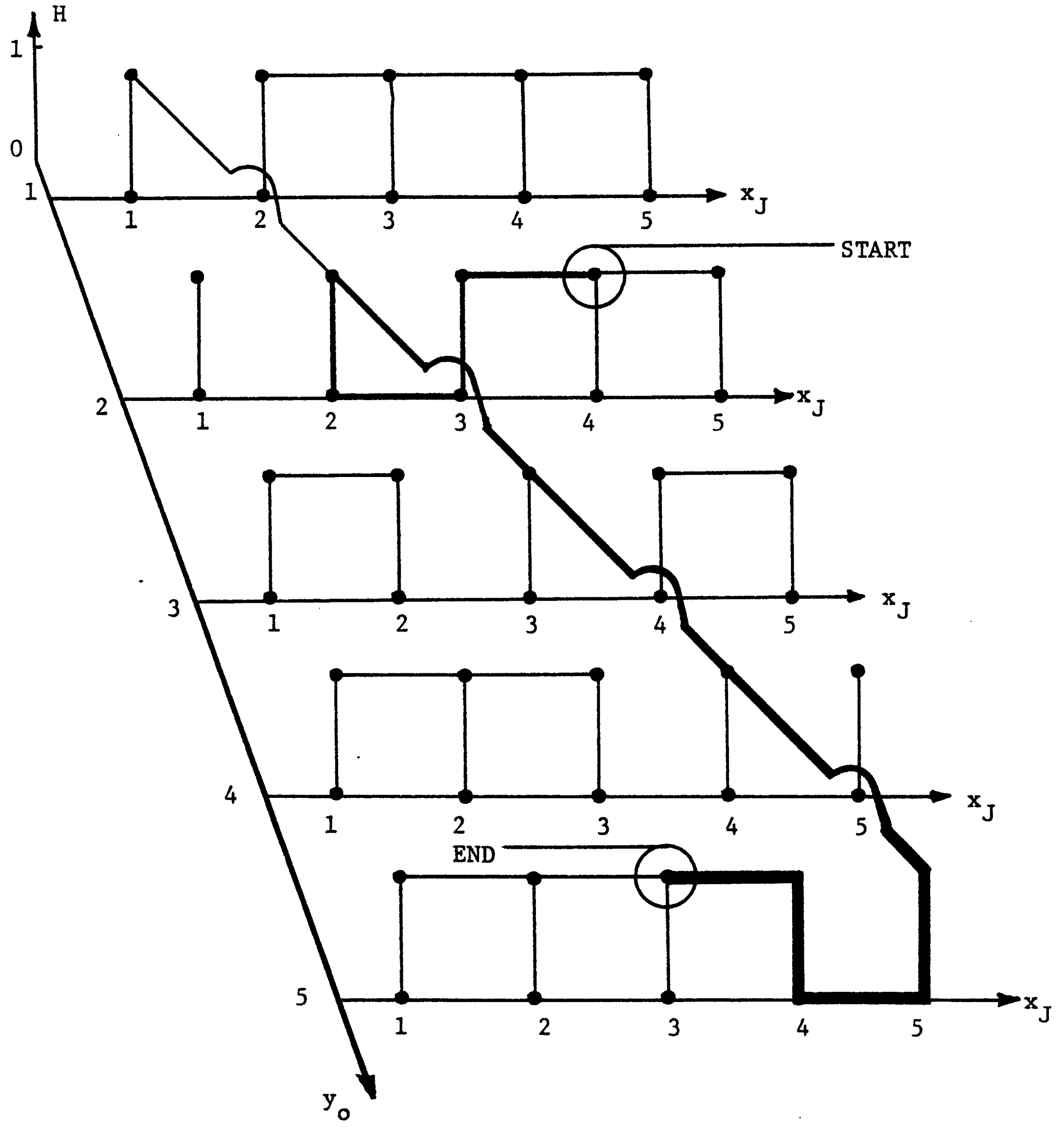


Figure 15

STATE SPACE SUITABLE FOR ALL CARRYING TASKS

Each segment of this Figure (corresponding to a single value of y_o) expresses all the constraints we denoted above regarding 1) dodging the object by straddling it, and 2) combining opening, moving and closing in the correct order to accomplish grasping. Each segment also shows, when compared to the whole, that the object cannot move by itself, while the jaws can. Taken as a whole, Figure 15 shows that the object can move only if the closed jaws coincide with it and they move together. Further, this cannot occur until after "Grasp" has been executed, nor can it terminate until "Release" has been executed. Thus the new state space encompasses all the constraints and information needed to plan moving and carrying tasks along a one dimensional physical space containing one pair of jaws and one object. Costs, though not shown, are consistent with Figure 11, with the cost of "Carry" being 4. This might reflect fuel cost because the object is heavy, or risk cost because it might get dropped, and so on. The path drawn between $\underline{S} = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix}$ and $\underline{S} = \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix}$ is sufficient in principle to execute in its entirety the following task: Starting with the jaws in 4, closed, and the object in 2, take the object to 5 and leave the jaws closed in 3. Reading the path, we obtain the following work description:

Move left one unit

Open

Move left one unit

Grasp

Carry right 3 units

Release

Move left one unit

Close

Move left one unit

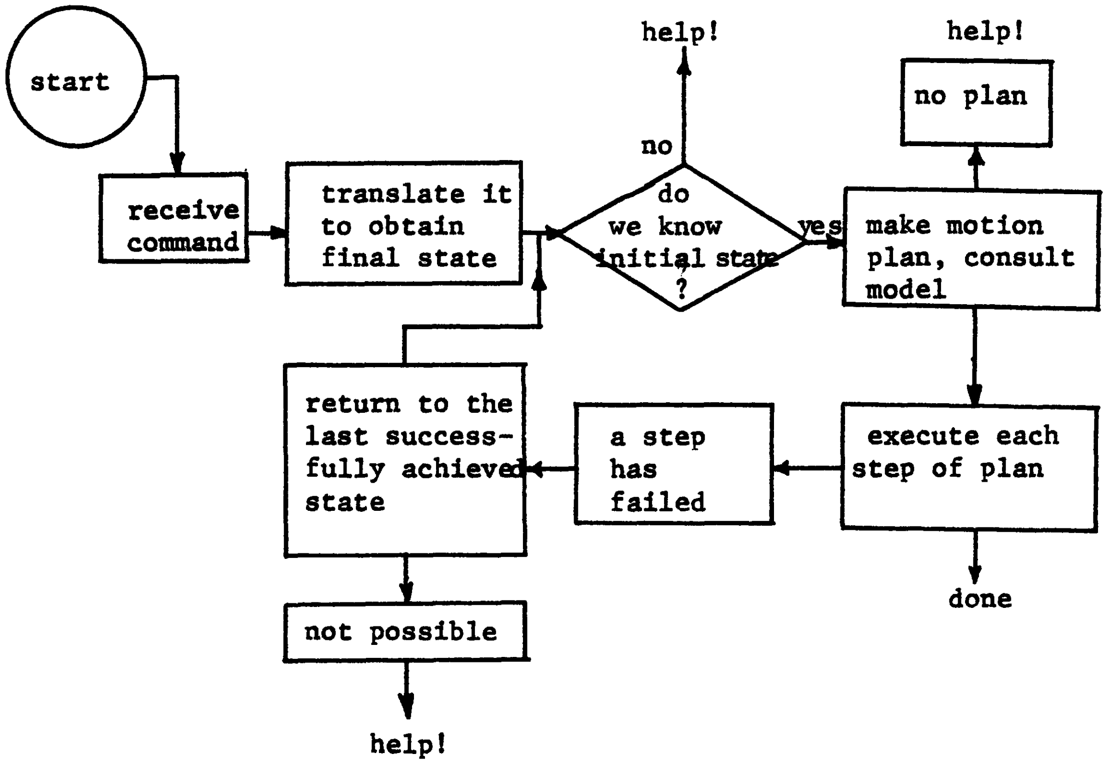
Done.

No matter where the object is in the space, any jaw motions will be planned either to straddle the object if its location is not to be changed (since moving it and replacing it would accomplish nothing and would cost more than necessary) or to grasp it if carrying is necessary to satisfy the operator's desire.

General Remarks

Note that, in all of the above examples, the paths carry the state through many intervening points on the way to the final value. These intermediate states can be thought of as subgoals to the operator's goal. Yet, at the manipulator primitive level, the intervening states are goals. Thus the conversion from the operator's task specification to the solution path consists of the replacement of a goal at or near the operator's level of concern with a string of goals closer to the manipulator's level of concern. Again, execution of this path plan must be supervised by an executive program similar to that sketched at the beginning of this chapter. The skeleton of this executive is easily deduced from the plan itself, except for the recovery procedure. The result is the following scheme:

DO A TASK


start
receive command
translate it to obtain final state
do we know initial state ?
no
help!
yes
make motion plan, consult model
no plan
help!
execute each step of plan
done
a step has failed
return to the last successfully achieved state
not possible
help!

This is a generalized version of the earlier program. The questions unanswered by both programs are the same: How to determine that a step has failed, how to return to the last successfully achieved state, how to determine whether or not this last state can be achieved, and so on. Some of these questions were studied by Ernst, many are being studied by Minsky.

It is at such points in the program that the operator might have to intervene. The system may have fouled up so badly that it cannot organize itself for the recovery: the object may have been dropped, or pushed aside, or smashed so that no amount of pawing about will locate it. The system may have damaged itself so that its sensors or effectors cannot function, or the effectors may be entangled in the environment and unable to move. When such things happen, the operator may be unable to rectify things using commands at the human primitive level. The language we have endowed him with is rich within its context, but the context is really quite limited. One cannot issue the command "Put the object back on the nearest quantized location point," or "Reassemble the object," or "Free the jaws from that tangle of pipes," because the translator has no points in the state space with which to represent the current disorganized condition of the environment or no commands with which to carry out the needed alterations to this condition. This is in spite of the fact that the operator may know full well what needs to be done. He just can't say it with the language we have given him, any more than a piano player can play notes that are in the cracks between the keys.

This language problem, which will be discussed more in a later chapter, is not the only problem facing designers of Supervisory Controlled Manipulator systems. Another difficulty is the amount of computing time or storage space required to do the calculations. Some aspects of this will arise regardless of the programming schemes used. In our case, the use of state space models threatens to demand huge amounts of storage space. The above examples indicate the extent of this: our physical space is only one dimensional with 5 points; to handle jaws and one object, we have five graphs of two dimensions each, totalling 50 points. (Figure 15). The same model, extended to cover a three dimensional space of only 10 points per axis, would contain 2×10^6 points. Attention to this problem, the "Curse of Dimensionality," will also be paid in a later chapter.

We summarize the results of this chapter as follows: A state space model of a manipulation task is a space representing alternative situations (positions or orientations of jaws and objects) which could be achieved using commands from a limited static set at the manipulator primitive level. Equivalently, since these commands can change the task site, the state space represents all the tasks which can be accomplished using arbitrary sequences of the allowed commands.

We demonstrated that the operator could express his desires in terms of new states he wished the system to occupy, and that this capability put the operator in control of the task site. We showed that state space models could express some non-trivial logical and physical constraints implicit in the successful execution of grasping, carrying, dodging obstacles, and so on. The operator's ability to influence the nature of the system's behavior by altering the values of the costs was also demonstrated. Finally, the task of deducing the correct sequence of commands to be used, or the correct sequence of subgoals to be achieved, was reduced to a shortest path problem.

In the next chapter, we shall consider several intriguing examples in which the ability of graphs to express order relations will be further exploited. The chapter after that will go into procedures by which the local computer finds shortest paths, and some of the interesting problems which arise when the state spaces get very large.

CHAPTER III

SOME EXAMPLES OF STATE SPACE MODELS OF MANIPULATION TASKS

In the previous chapter, we saw how graphs can store the logical and physical constraints required to perform basic obstacle avoidance, grasping, and carrying. In this chapter, we consider six non-trivial examples in which various properties of manipulation are expressed. They are

1) Simple decision-making
2) Pushing an object with the jaws
3) Pushing an object with another object
4) Maneuvering an object through a crowded environment
5) Manipulating with two pairs of jaws
6) Complex ordering and decision-making

1) As an example of simple decision-making, consider Figure 1. A square object and a round object lie on a table which has five quantized locations. The square object is in location 1, while the round object is in location 3:

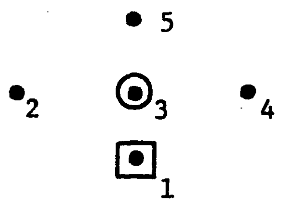


Figure 1

THE DODGING PROBLEM

We want to move the square object to location 5. Is it better to move the square object around the round object (via locations 2 or 4) or move the round object out of the way and then move the square object directly to location 5 via location 3? Much depends, of course, on what we mean by "better."

The decision problem involved here can lead to very large and complex graphs, because the manipulator jaws do not move empty or grasp free of cost. As a result, a complete solution would require a state vector containing the locations of each object plus jaws, for each of the two values of the jaw status variable H. Thus the graph would consist of two connected subgraphs, one for each value of H, each having three dimensions with 5 x 5 x 5 points. Since we can't draw this easily, let us assume that empty jaw movements are essentially free, and graph instead only the positions of the two objects. The result is complex enough. The state vector consists of the location number of each object, giving us a two dimensional state space. See Figure 2. A point $\begin{bmatrix} i \\ j \end{bmatrix}$ in this space indicates that the round object is in location i while the square object is in location j. Each horizontal edge represents moving the round object while holding the square object fixed, and the reverse for each vertical edge. The diagonal vertices $\begin{bmatrix} i \\ i \end{bmatrix}$ are forbidden since both objects cannot occupy the same location simultaneously. Despite its complexity, this graph, like all the others we consider, can be generated solely by considering the physical possibilities at each state, quite without regard for what can be done at any other state. We need never map out or try to comprehend all combinations of moves at once. The graph takes care of that for us.

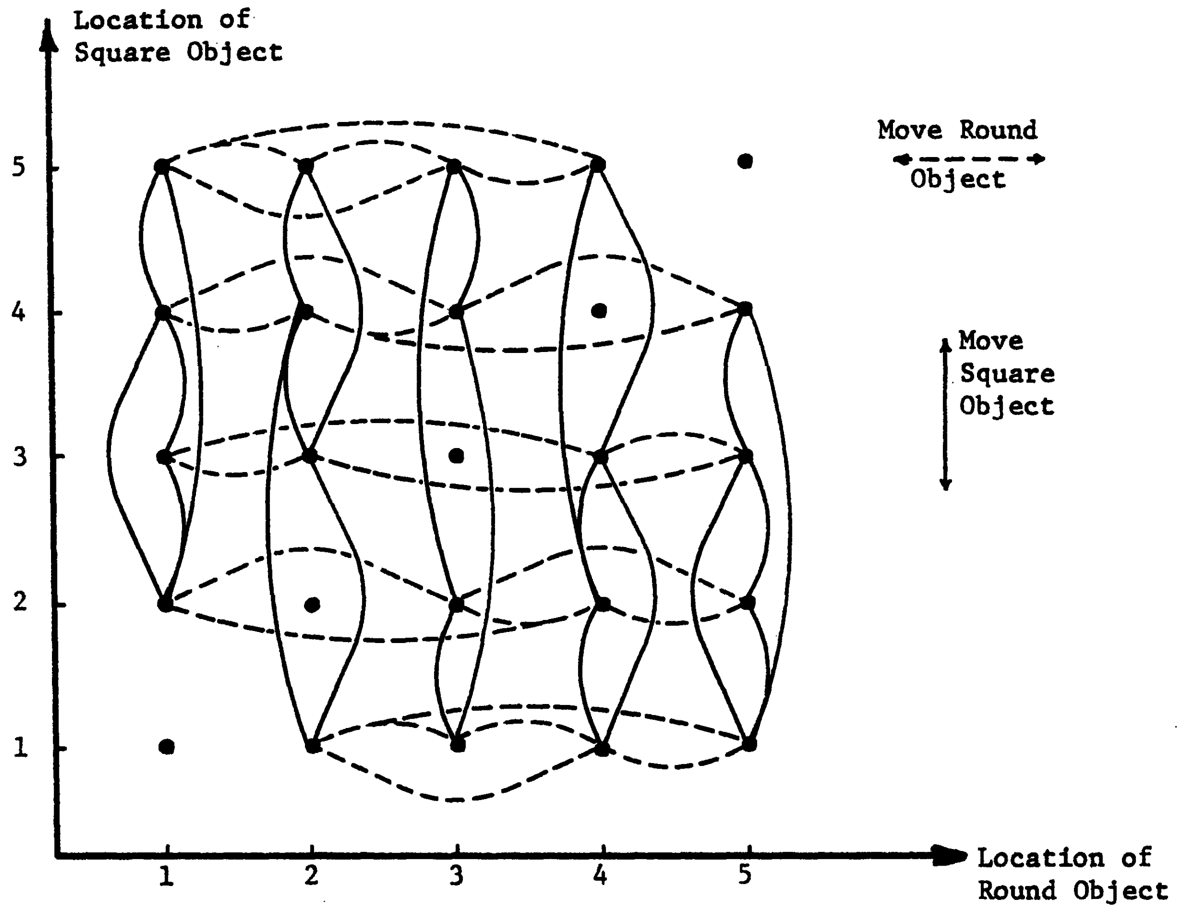


Figure 2

STATE SPACE FOR DODGING PROBLEM. SOLID EDGES SHOW MOVES OF SQUARE OBJECT. DOTTED EDGES SHOW MOVES OF ROUND OBJECT.

The system shown in Figure 1 then occupies state $\begin{bmatrix}3\\1\end{bmatrix}$, and any of the states $\begin{bmatrix}1\\5\end{bmatrix}$, $\begin{bmatrix}2\\5\end{bmatrix}$, $\begin{bmatrix}3\\5\end{bmatrix}$, or $\begin{bmatrix}4\\5\end{bmatrix}$ will suffice as the terminal state, so far as the operator is concerned.

Costs might be assigned in a variety of ways. The square object might be heavier than the round, or the round harder to grasp reliably than the square. Either consideration would cause all moves of one object to be higher in cost than moves of the other. Or, moves like 1-2 are longer than moves like 1-3. Yet again, it might be felt that moving the square object into 2 or 4 while the round object is close by in 3 is risky due to close clearance, and would be less costly in risk if the square object were moved to 2 only after the round one has been shifted to 4, or vice versa. These last two considerations would make some moves more costly on the basis of what states they join, rather than what object is being moved. We shall say no more about this example, since the interesting points have been made.

2) The next example is that of using the jaws to push an object. This we develop quickly using Figures 8 and 15 of Chapter II. The jaws can push the object to the right if $y_o - x_J = 1$, and can push the object to the left if $y_o - x_J = -1$, each relation implying that the jaws are adjacent to the object on the appropriate side. The system must be made to understand that, during pushing, the jaws and object both move so that the relation $y_o - x_J = \pm 1$ (as the case may be) continues to be satisfied all the time. In addition, the system must understand that the object cannot be pulled to the left when $y_o - x_J = 1$ or to the right when $y_o - x_J = -1$. We accomplish all this by connecting points obeying $y_o - x_J = 1$ with directed arcs labelled "Push right," so that the arrows allow x_J and y_o only to increase. Points obeying $y_o - x_J = -1$ are connected by arcs labelled "Push left," with the arrow indicating that x_J and y_o are allowed only to decrease. The result is Figure 3.

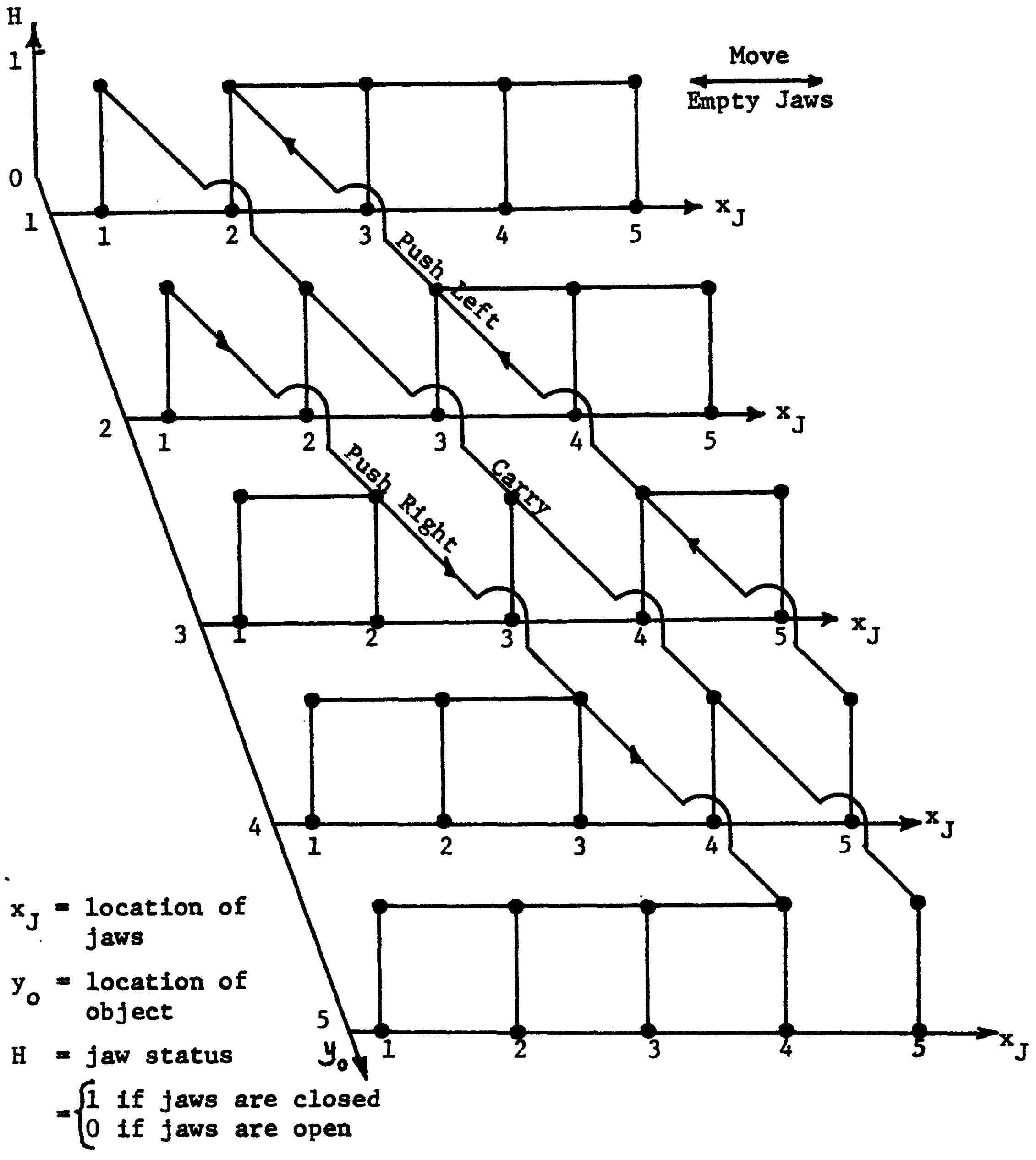


Figure 3

STATE SPACE ALLOWING JAWS TO PUSH THE OBJECT

Note that to push right and then push left, the jaws must, after pushing right, first open, straddle the object to get to the other side, then close and initiate pushing left. The graph also shows that if the object is in 5, say, then it cannot be pushed right at all, nor pushed left until it has been carried at least to location 4, since there is no room to the object's right in which to put the jaws. So these six new lines add a lot of information. This is our first example of a mixed graph model of a manipulation task.

3) The next example is that of using one object to push another object into a hole. (Pushing directly by the jaws is not allowed.) The physical space is shown in Figure 4.

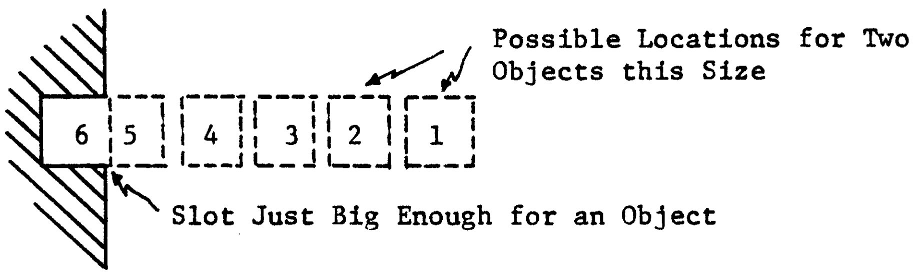

Possible Locations for Two Objects this Size
6
5
4
3
2
1
Slot Just Big Enough for an Object

Figure 4

PHYSICAL SPACE FOR PUSHING PROBLEM

There are two objects, _a_ and _b_, _a_ being nearer the slot than _b_. The size of the slot demands that _a_ be pushed into the slot, with _b_ acting as pusher, held in turn by the jaws. The motion of the objects is our major concern, so we ignore the motion of the jaws for the time being. The state vector is then

$$\underline{S} = \begin{bmatrix} x \\ \\ y \end{bmatrix} \qquad \begin{array}{l} x = \text{location of } \underline{a}, = 1, \ldots, 6 \\ \\ y = \text{location of } \underline{b}, = 1, \ldots, 6 \end{array}$$

and the corresponding state space is shown in Figure 5, with the allowed commands written on the appropriate arcs and edges.

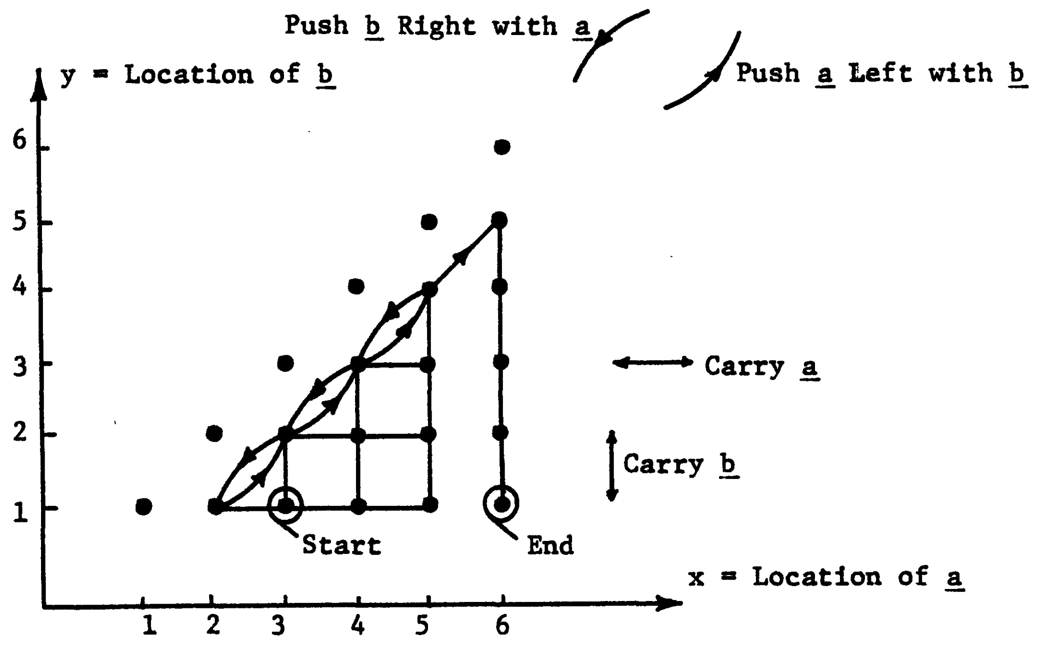


Figure 5

STATE SPACE FOR THE PUSHING PROBLEM

The arcs along the sub-diagonal are not condensed to edges in order to show that certain pushing is allowed but no pulling.

Say a is in location 3 and b is in location 1, and we ask for a to be put into the slot and b to end up in location 1. Then the initial state is $\begin{bmatrix}3\\1\end{bmatrix}$ and the final state is $\begin{bmatrix}6\\1\end{bmatrix}$. It is clear that the system will "figure out" that b must push a into the slot in order that the task be accomplished. If we decide that pushing is more risky than carrying (the pushed object might fall away to one side), we can charge more for pushing than for carrying. Then the resulting path, regardless of its other characteristics, will contain exactly one "Push," the required one from $\begin{bmatrix}5\\4\end{bmatrix}$ to $\begin{bmatrix}6\\5\end{bmatrix}$. The "other characteristics," however, could be very unsatisfactory. If carrying costs the same for a and b, while empty jaw motions are free, then zig-zag paths from $\begin{bmatrix}3\\1\end{bmatrix}$ to $\begin{bmatrix}5\\4\end{bmatrix}$ are as cheap as any. They imply that a is carried part way to 5, then b carried part way to 4, then a carried a little farther, then b a little farther, until finally a is in 5 and b is in 4, ready for pushing. This kind of behavior can be eliminated by charging for motions of the empty jaws. This requires that the jaws' position and H be added to the state vector. Then a minimum cost path would consist of carrying a directly to 5, carrying b directly to 4, then pushing a into 6 and carrying b back to 1.

4) In the following example, we maneuver a long thin spar through a crowded environment. Here the interactions between the environment and the spar's position and orientation are crucial at each step. Since only one object is involved, we again ignore the jaws and take the state vector to be

$$\underline{S} = \begin{bmatrix} x \\ y \\ \alpha \end{bmatrix} \quad \begin{array}{l} x = x \text{ coordinate of object} \\ y = y \text{ coordinate of object} \\ \alpha = \begin{cases} 0 \text{ if object is parallel to x axis} \\ 1 \text{ if object is parallel to y axis} \end{cases} \end{array}$$

The usual carry commands change x and y, while a command called "Rotate" changes α. Thus both position and orientation of the spar are quantized. The physical space is shown in Figure 6. Walls are shown as open rectangles, while the two possible orientations of the spar are shown by cross lines at each possible object position.

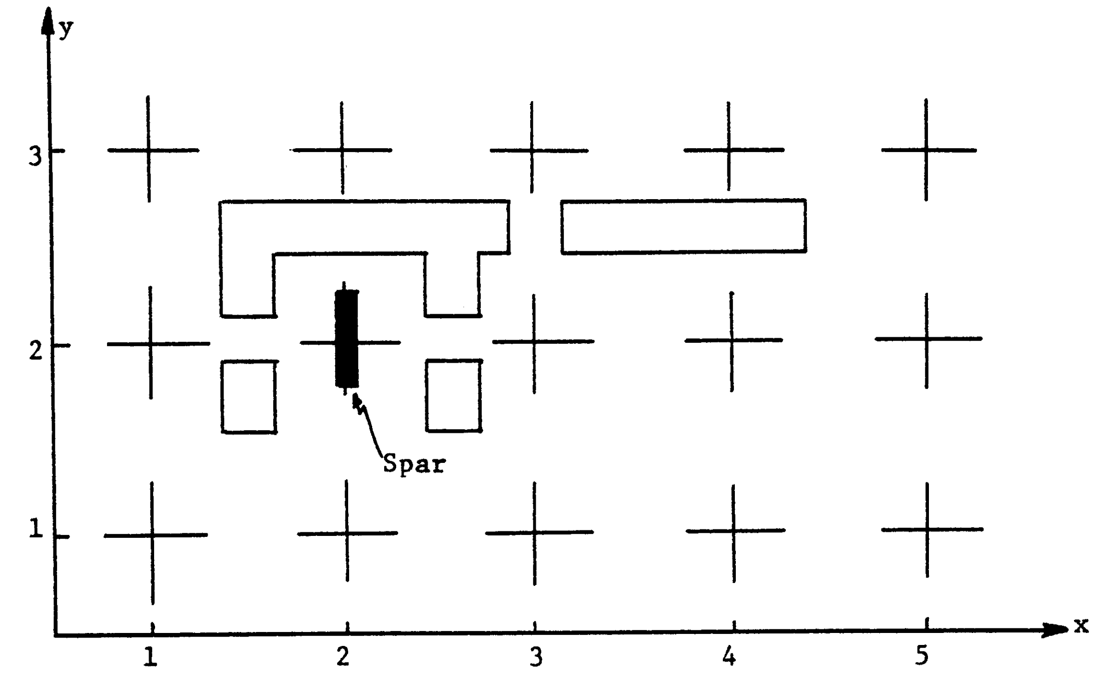


Figure 6

PHYSICAL SPACE FOR THE SPAR PROBLEM

The challenge is provided by the doorways, which allow the spar and jaws to pass axially but not athwart. The jaws can open wide enough to grasp the object only one way. See Figure 7.

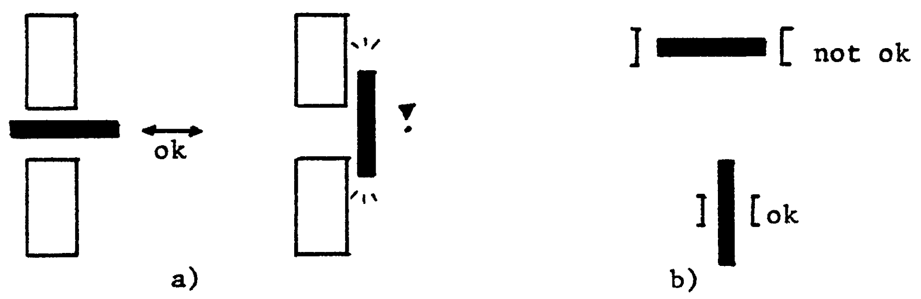


Figure 7

a) DOORWAYS AND PASSAGE OF A THIN OBJECT. b) GRASPING THE OBJECT IN ONLY ONE ORIENTATION

We assume that the object is in the jaws at the beginning and end of the task.

A good way to visualize the graph associated with this problem is to put all values of $\underline{S}$ for which $\alpha = 0$ on one plane, and all values of $\underline{S}$ for which $\alpha = 1$ on another plane, drawn so that it appears to lie behind the first plane. We assume for illustration that all carries cost 2 while rotations cost 3. Let the object be at location (2, 2), with orientation parallel to the y axis, and say we want it moved to location (3, 3) and end up oriented parallel to the x axis. Then the initial state is $\begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$ and the final state is $\begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix}$. The resulting graph appears in Figure 8, while two solution paths of equal cost appear in Figure 9. The solution path is visualized on a sketch of the environment in Figure 10. An interesting feature of these paths is that they do not "look like" the most direct route. This is a feature we get used to seeing in optimal path solutions. Closer examination reveals that the optimal paths, by moving the object away from the desired final state, are able to save two rotations by spending a little more distance. Again, if we read the path, we get a list of the required carries and rotates in the correct order. A more general solution to this problem is discussed in Chapter V.

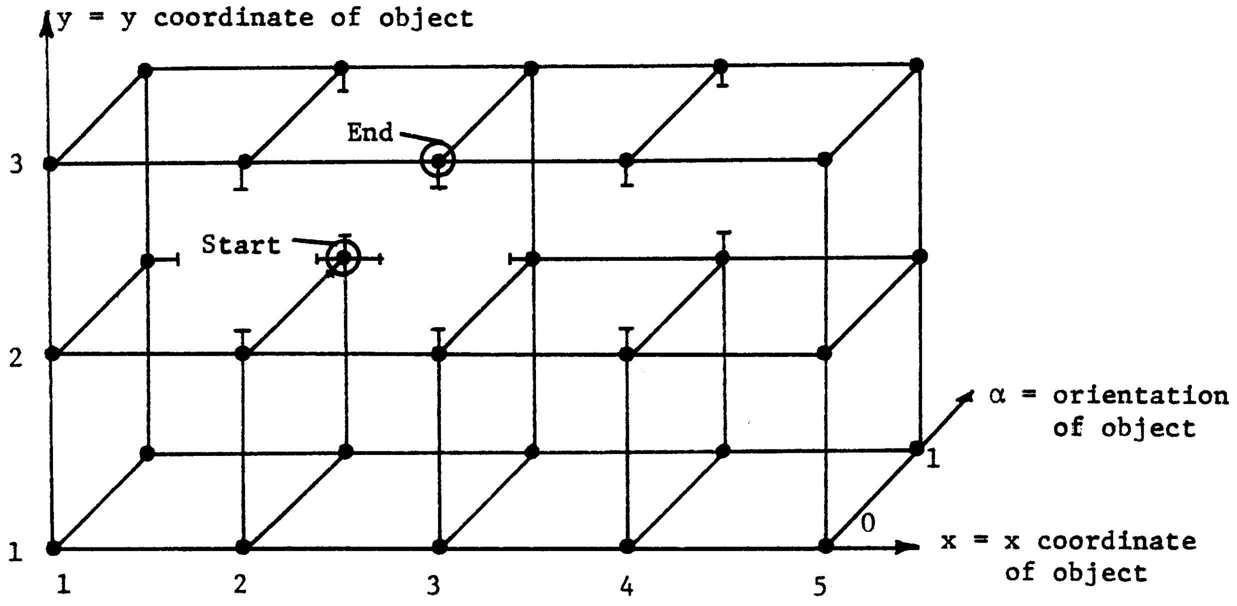

y = y coordinate of object
3
2
1
1
2
3
4
5
End
Start
α = orientation of object
1
0
x = x coordinate of object

Figure 8

GRAPH OF THE PROBLEM IN FIGURE 6

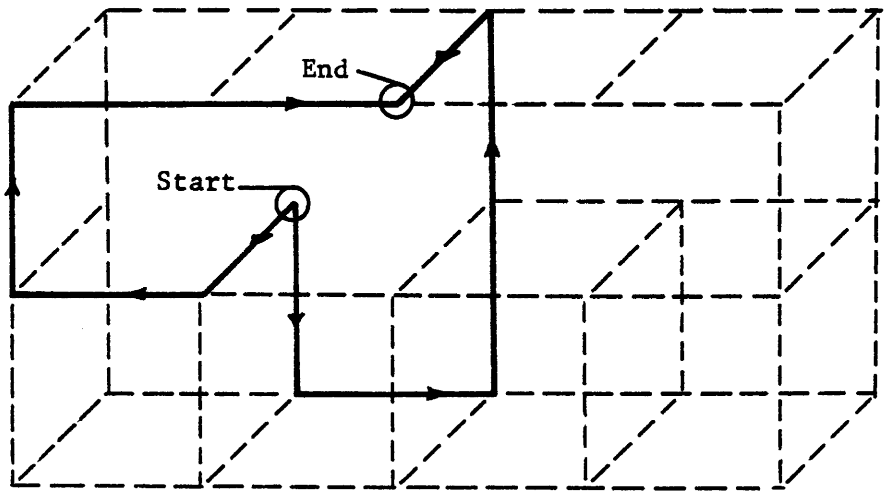


Figure 9

SOLUTION TO THE GRAPH OF FIGURE 8

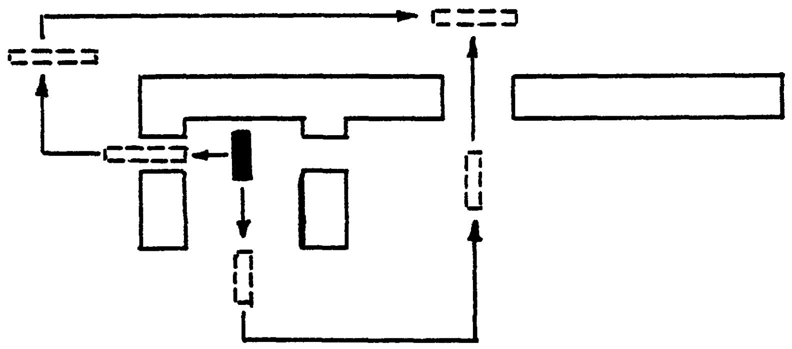


Figure 10

VISUALIZATION OF THE SOLUTION PATHS FROM FIGURE 9

5) The next example concerns manipulating one object with two sets of jaws in a one dimensional physical space. It appears in Greene (op. cit.), where it is used to illustrate some of the problems posed in modelling the sensorimotor behavior of infants. Greene does not solve the example explicitly, but uses a continuous space similar to state spaces described here to indicate the nature of the solution.

The physical space is shown in Figure 11. The possible locations for the jaws and object are x = -2, -1, 0, 1, 2. The left and right jaws have limited ranges, sharing only location x = 0. We take the state vector to be

$$\underline{S} = \begin{bmatrix} x_o \\ L \\ R \\ H \end{bmatrix} \quad \begin{array}{l} x_o = \text{location of object, } = -2, -1, 0, 1, 2. \\ L = \text{location of left jaws, } = -2, -1, 0 \\ R = \text{location of right jaws, } = 0, 1, 2 \\ H = \text{jaw status variable, as before} \end{array}$$

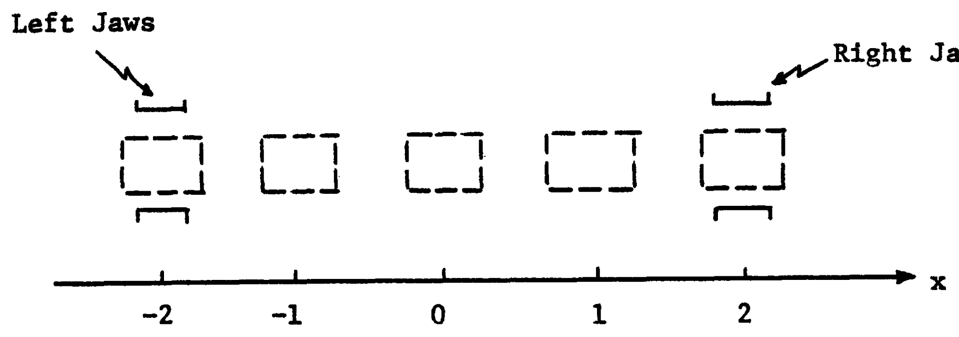


Figure 11

PHYSICAL SPACE FOR TWO JAWS PROBLEM

and we allow commands

Open/Close left jaws

Open/Close right jaws

Move left jaws left/right one unit

Move right jaws left/right one unit

Change jaws in use from left to right or right to left.

The resulting state space would be very complex except that we impose some simplifying conditions:

a) Only one set of jaws moves at a time

b) The jaws not in use occupy location 1 or -1 as appropriate

c) Both jaws cannot occupy location 0 simultaneously

The resulting state space appears as Figure 12. The command "Change jaws in use" is indicated by the dotted line. It makes no change in the environment, but it is not free and is a significant change in emphasis. It serves only to alter the local computer's

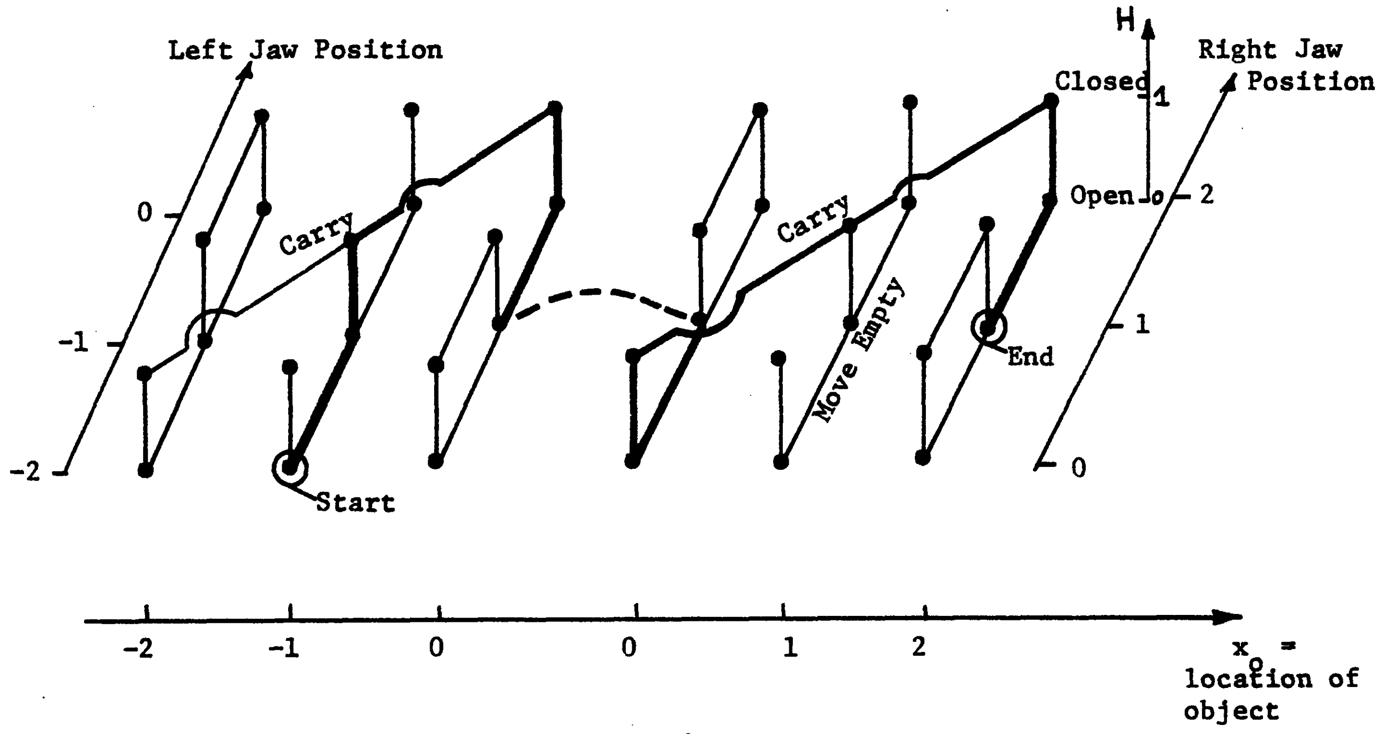


Figure 12

STATE SPACE FOR TWO JAWS PROBLEM

point of view of the problem and thus is unique among the commands we have discussed so far. The task: Move the object from location -1 to +2, leaving the right jaws in +1, may be specified by the states marked Start and End in Figure 12. The resulting path, assuming costs similar to those of previous examples, is also shown. The left jaws carry the object to location 0, then retreat to location -1. The right jaws then move into location 0, pick up the object and carry it to location 2. The right jaws then go to location 1.

Similar techniques would allow us to model a problem in which the object could be passed directly from one pair of jaws to the other, provided either that each pair assumes the proper orientation or that the object is long and thin.

6) The last example we call the blocked doorway problem, about which we shall have more to say in Chapter V. Here we develop the straight-forward solution using techniques which are by now familiar. On a two dimensional table is a wall with a narrow doorway. The door is blocked by a moveable object, B. We wish to move another object, A, through the door to a location, X, on the other side. We have at our disposal a pair of non-rotating jaws which move in the plane only. See Figure 13.

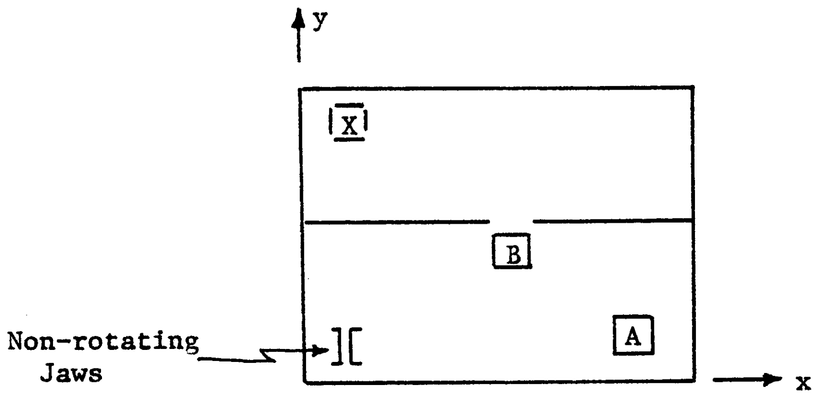


Figure 13

PHYSICAL SPACE FOR THE BLOCKED DOORWAY PROBLEM

A graph of seven dimensions (x, y position of each object plus jaws, and H) will allow a formal solution to this problem. We merely ask for the path which changes the state vector

$$\underline{S} = \begin{bmatrix} x_A \\ y_A \\ x_B \\ y_B \\ x_J \\ y_J \\ H \end{bmatrix}$$

x_A, y_A = x,y coordinates of object A

x_B,y_B = x,y coordinates of object B

x_J,y_J = x,y coordinates of jaws

H = jaw status variable as before

so that (x_A, y_A) coincide with the coordinates of location X. The result is that the jaws will go to B, grasp and carry it to one side and leave it optimally with respect to the next moves: the jaws go to A, grasp and carry it to X, then return to B and replace it in front of the door, then return to their original position. Object B and the jaws end up at their initial positions precisely because the state transition we requested involved only a change in the location of A. As is standard in optimality problems, we get what we ask for. The operator is free, of course, to pick some other end condition, such as B to the right of A and jaws at the door, or many others corresponding to a variety of tasks.

This solution is fairly impressive. The operator need not have known that an object blocked the door. He merely asked that A be moved to X and that is what he would get. If more than one object blocked the door, a state space which recognized all of them would deduce the optimal order in which to move each, and the optimal location for each, in order to unblock the door.

Of course, most of this is the merest flight of fancy, for the size of the resulting spaces would be phenomenal. The original seven dimensional space would have $2 \times N^6$ points, N being the number of points on each of the x and y axes. Even if N were 10, the result would be too big to be of practical use. Each additional object in the state vector would add two to the exponent of N. This is another obvious place at which to call in the operator. Clearly, much of the space in the graph would be devoted to expressing motion possibilities for B which will be rejected immediately, such as moving it to X, or to the far corner opposite X. Many of the connections which are mathematically conceivable are not physically sensible, those which express moving both objects at once, or moving an object without moving the jaws, connections which will therefore be absent. Think of the saving if a state space like that of Figure 3 could be used: let the operator factor the task into "Move B to Y," plus "Move A to X." Then only one object need be considered moveable at a time, the other being merely an obstacle, off limits to jaws and the moveable object. Then a graph of 2 x N^4 points suffices, a significant improvement if only because it is practical. But it is more than just practical.

It is the operator who can easily discern that B, while subordinate to the main task, still must be dealt with first. Such a concept _can_ be stored in the appropriate graph, but it is wasteful of space and computing time: the computer is overburdened with doing the part which is _easy_ for the operator, diverting its resources from doing the dirty work of planning detailed motion, the part which is _hard_ for the operator. Thus such a state vector mismatches the man and the machine. In Chapter V we shall develop some methods for involving the operator more fully when subgoals must be identified. For now, the emphasis is on what concepts can be embodied in a graph.

CHAPTER IV

SHORTEST PATH PROBLEMS

In this chapter, we take up mechanistic methods by which shortest paths are found in graphs. Many of these methods are closely linked with Dynamic Programming and the Principle of Optimality.[4] The application of these methods to large graphs presents special problems, some of which will be dealt with below.

We are interested in paths between two states in a graph and in how long these paths are (or how much they cost). Define the distance between states A and B by D(A,B), such that if a path connects A to C via B, then, along that path,

$$D(A,C) = D(A,B) + D(B,C) \tag{1}$$

If there is an arc from A to B, then denote the distance from A to B along this arc by d_{AB}. Assume $d_{AB} > 0$. If there is another arc from B to A, then in general $d_{AB} \neq d_{BA}$, except when the arcs are condensed into an edge.

Let $D^*(A,C)$ be the shortest distance from A to C. How do we find this distance and the corresponding path? Start at A. Then $D(A,A) = D^*(A,A) = d_{AA} = 0$. Next, look at <u>all</u> the B_i adjacent to A. See Figure 1.

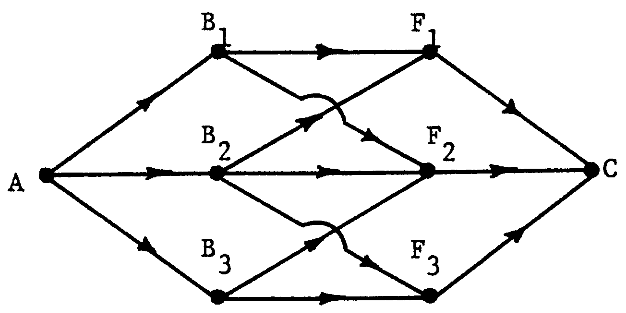


Figure 1

EXAMPLE GRAPH

Then, for each i, $D^*(A,B_i) = d_{AB_i}$. Next, look at each F_j adjacent to each B_i. The shortest distance from A to each F_j minimizes the sum $d_{AB_i} + d_{B_iF_j}$ or

$$D^*(A,F_j) = \min_i \left[d_{AB_i} + d_{B_iF_j} \right] \tag{2}$$

At each F_j, there will be a best i. Call it i*/j, meaning the best i for that j. Next, look at C. The shortest distance from A to C satisfies

$$D^*(A,C) = \min_{i,j} \left[d_{F_jC} + (d_{B_iF_j} + d_{AB_i}) \right] \tag{3}$$

$$= \min_j \left[d_{F_jC} + \min_{i/j} (d_{B_iF_j} + d_{AB_i}) \right] \tag{4}$$

where $\min_{i/j}$ means minimum over i for a given j. But, using (2), we can rewrite this as

$$D^*(A,C) = \min_j \left[d_{F_j C} + D^*(A,F_j) \right] \tag{5}$$

As before, denote the best j by j*. Then the cheapest way to arrive at C is via F_{j*} and the cheapest way to arrive at F_{j*} was found by equation (2) to be i*/j*. Thus the best path to C is found by backtracking through the starred subscripts to A. Since each edge corresponds to the execution of a particular command, we obtain a list of the required commands implicit in the list of starred subscripts. Then we are ready to execute the task. Equation (5) is a version of the functional equation of Dynamic Programming, which is used in various forms to solve a wide variety of optimality problems.

The important thing about (5) is that C could be any node on the graph. To find $D^*(A,\alpha)$, we need only know $D^*(A,\beta_i)$ at all the β_i adjacent to α. This suggests that $D^*(A,\alpha)$ is a function of the argument α. To evaluate it, we work our way out in some fashion from A, the node at which D*(A,A) is known to be zero, a sort of boundary value. Because the function D* can be so compactly defined in terms of its predecessors (really in terms of itself), we are spared the task of writing longer and longer strings of simultaneous minimizations like (3) as the paths get longer. All the rest of the path, and all the rest of the minimization, is abbreviated in the symbol D* which appears on the right in (5).

A function defined in terms of itself, such as D*, is called <u>recursive</u> or <u>computable</u>. A familiar example is the factorial function:

$$\text{factorial }(n) = n \cdot \text{factorial }(n-1) \qquad n > 0 \tag{6a}$$

$$\text{factorial }(0) = 1 \tag{6b}$$

Equations like (6) can readily be used to find factorial (n) without storing a table of values. We work our way out, in an obvious order, from $n = 0$, using (6a) again and again. Thus (6a) enables one to compute n! for any integer $n > 0$, just as (5) enables one to compute $D^*(A,\alpha)$ for any node α. The main virtues of recursive functions are their compactness and the uniformity of the calculation method regardless of the value of the argument.

The Principle of Optimality says: If the shortest path from A to C passes through F_j just before reaching C, then the portion of that path joining A and F_j is in fact the shortest path from A to F_j. If it were not, then a shorter path from A to C could be found. Equation (5) expresses this by telling us that to find the shortest path from A to C, we must first find all the shortest paths from A to each of the F_j adjacent to C. Thus the Principle of Optimality and (5) express the same property of shortest paths.

The length of the total path, furthermore, is stationary with respect to integral changes in j exactly when $j = j^*$. This means that j^* minimizes the total path length, not merely the immediate length d_{F_jC}. Thus (5) is capable of detecting the optimal path even when that path must suffer a relative loss initially in order to achieve ultimate improvement. The paths in Example 4 of Chapter III have this property.

The trouble with (5) is that, unlike (6), there is no obvious order in which to work our way out from node A on the general mixed graph. This difficulty becomes obvious if one allows the B_i or the F_j to be connected together in Figure 1, or if the arrows are removed. Then there is no definite direction "out" from A. Some of the methods of finding shortest paths face this difficulty directly, but many do not. The latter tend to be simpler but less efficient, since they investigate many ways out which have no chance of being optimal.

Algorithms

Bellman[3] proposed an algorithm in which one first finds the shortest one-arc path (if any) from the starting node to each other node, then the shortest path of two or less arcs, using the previously generated one-arc paths. Next one finds the shortest path of three or less arcs, again using all the previously generated data. Because one successively improves the path, this recursive method may be called Path Iteration.

A sort of dual of Path Iteration is called Value Iteration, in which one successively improves deliberately pessimistic initial estimates of the distances from A to each other node α. Such methods are also called Labelling Algorithms. One of the first, and by far the simplest to implement, is Ford's algorithm.[13] A clear exposition of it may be found in Berge (op. cit.). A brief description follows:

Label the initial node A with an index D(A,A) equal to zero, and label each other node α with an index $D(A,\alpha) = \infty$. Select any adjacent pair of nodes, α and β.

$$\text{If } D(A,\alpha) - D(A,\beta) > d_{\beta\alpha} \tag{7a}$$

then reduce $D(A,\alpha)$ to the value

$$D(A,\alpha) = d_{\beta\alpha} + D(A,\beta) \tag{7b}$$

and indicate at α that β is the node associated with this reduction in $D(A,\alpha)$. Otherwise, do not change $D(A,\alpha)$. When this procedure fails to achieve a reduction in any index, the work terminates. $D(A,\alpha)$ is then $D^*(A,\alpha)$, and the optimal path may be read off in reverse order as in Dynamic Programming. To see why this is true, consider applying (7) to all the β_i adjacent to α. Then, unless all the $D(A,\beta_i) = \infty$, (7) is equivalent to*

$$D(A,\alpha) = \min_i \left[d_{\beta_i\alpha} + D(A,\ \beta_i) \right] \tag{8}$$

with

$$D(A,A) = 0$$

If we start by taking α to be each of the nodes adjacent to A, then it is clear that Ford's algorithm is merely forcing the indices $D(A,\alpha)$ to satisfy the Principle of Optimality (Equation (5)) by successive approximations and that this algorithm is just another way of employing Dynamic Programming. Note that, as in the Path

---

* This equivalence breaks down if all the adjacent $D(A,\beta_i) = \infty$, since in that case, $D(A,\alpha)$ will not undergo any change, in accordance with the algorithm. If this condition persists to the end of processing, then this α and the β_i are isolated from A.

Iteration algorithm, there is no way of specifying both a starting and a terminal node. As a result, Dynamic Programming finds all the shortest paths from A to each α in the graph.*

However, there is no clue as to the order in which the α's should be selected for use in (7). Merely selecting each α in the graph once in an arbitrary sequence will not do, as is shown by the example illustrated in Figure 2. Each edge is labelled with its cost:

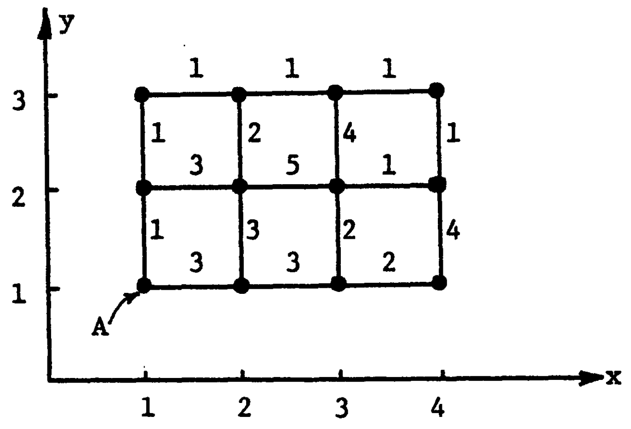


Figure 2

EXAMPLE GRAPH

If we arbitrarily select the α's by starting at (1, 1) and moving left to right across each row, we obtain the following set of D(A,α)'s, together with arrows which point from each α to the β associated with the last reduction of D(A,α). See Figure 3.

* We could as well designate the terminal state and find all optimal paths which lead to it.

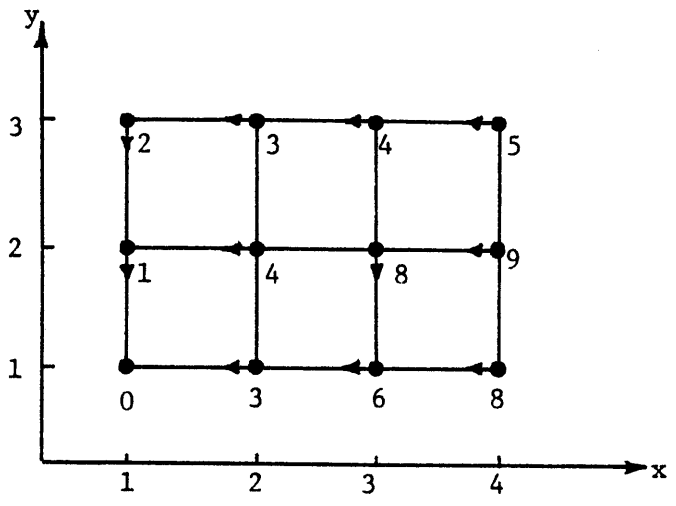


Figure 3

NUMERICAL EXAMPLE, PHASE I

Thus, after making one thorough pass over the graph, it might appear that the shortest path from (1, 1) to (3, 2) is 8 units long and runs from (1, 1) horizontally to (3, 1), thence vertically to (3, 2). (Paths and lengths to other nodes also appear.) But this is clearly wrong. One more thorough pass by rows changes D(A, (4,2)) from 9 to 6, with its arrow pointing to (4, 3), while another such pass changes D(A, (3,2)) from 8 to 7. A fourth pass makes no change, leaving us with the arrows from (3, 2) to A as shown in Figure 4:

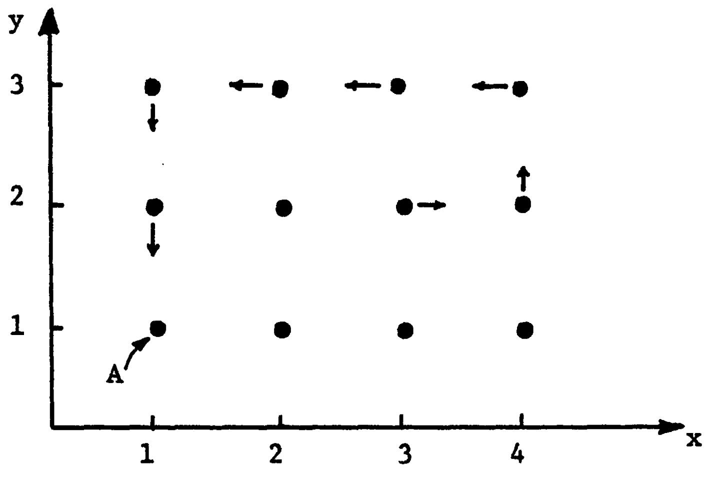


Figure 4

NUMERICAL EXAMPLE, FINAL PHASE

Thus we must keep passing over the graph until the D's stop changing.

We can redraw Figure 2 so that it looks like Figure 1, that is, so that a necessary sequence for choosing the α's emerges unambiguously. To do this, we basically add "time" to the state vector. At "time" zero, we are at A. Draw A at the left. Draw all the other 11 nodes in a column immediately to A's right. Connect A by arcs to those nodes in this column which are adjacent to A in the original graph, but do not connect any of the column members to each other. Label each connection with its cost from the graph. This column corresponds to "time" one. To the right of this column, draw these 11 nodes again and connect a node α in column one by an arc to a node β in column two if β is adjacent to α in the graph. Label the

The nodes in each row correspond to one node in Figure 2, as shown.

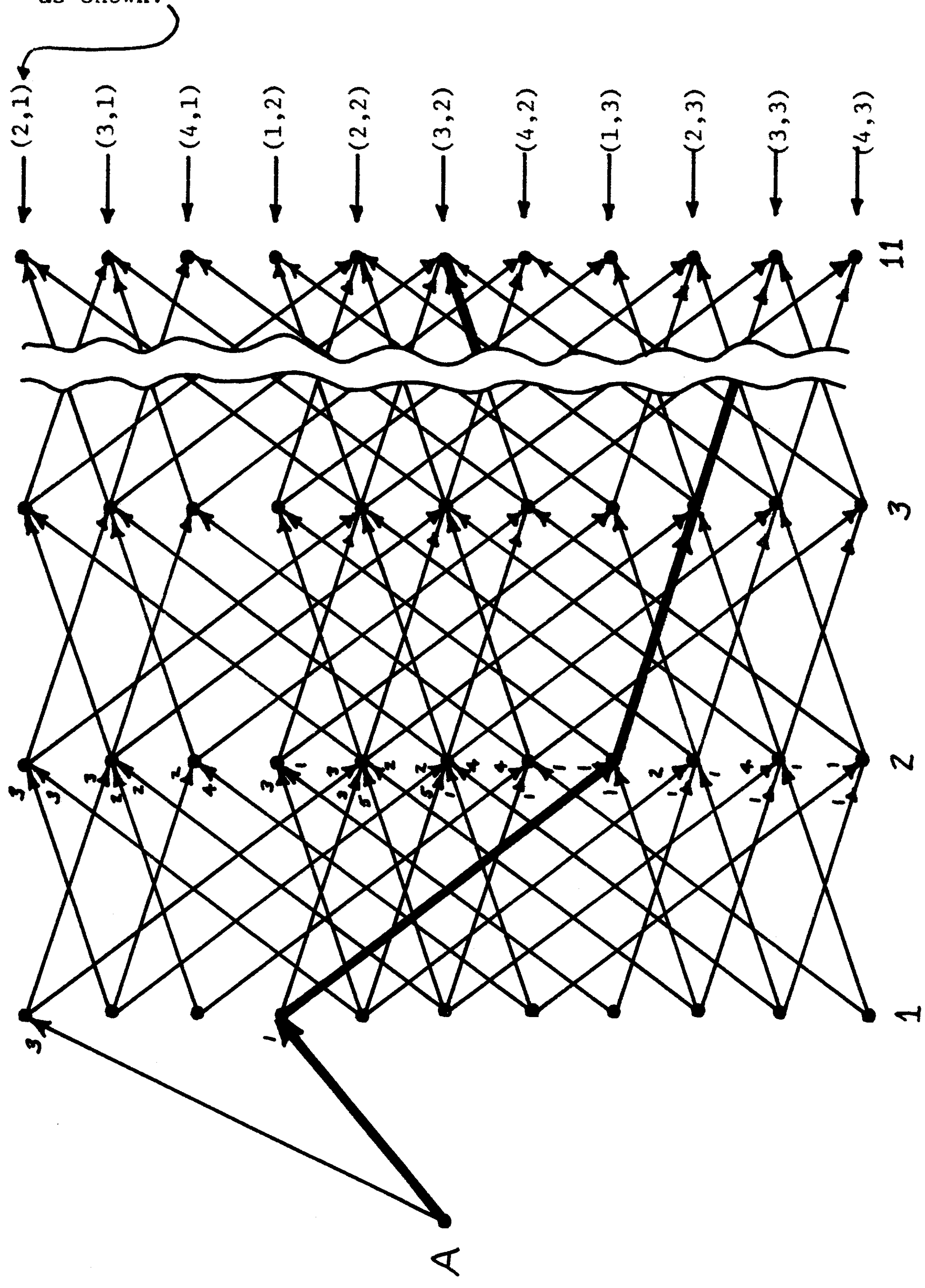


Figure 5

COLUMN GRAPH CORRESPONDING TO FIGURE 2. FINAL PATH SHOWN IN HEAVY LINES.

corresponding cost. Do this until there are 11 columns. The result is shown in Figure 5. Paths from A are then traced left to right, touching the columns once each in order. No more columns are needed since no optimal path in this graph can have more than 11 edges.

Then we apply equation (5) from left to right, choosing for C each node in the first column (in no particular order), then each node in the second column, and so on. This is equivalent to 11 thorough passes over the original graph and is fortunately emenable to a certain reduction in rapid access memory requirement.[18] Using a construction like Figure 5, we can show that in a graph of N nodes, no more than N - 1 thorough passes of the Ford algorithm are needed to reduce the D's to their optimal values and find the optimal paths.

However, the above column method demands N - 1 passes (plus storage in low speed memory for N - 1 times more points) while the Ford algorithm will stop anytime a pass reveals no index changes, which typically occurs well before the N-1st pass. So the column method solves the α selection problem in a most inefficient way. It is included here to show the connection to more conventional Dynamic Programming, in which there is often a variable which, like time, is "used up" in some sense and can be employed to show which direction is "away" from the initial state.

The advantages of the Ford algorithm are its simplicity and relatively low storage requirements. (At each node α, only $D(A,\alpha)$, the arrow, a list of the adjacent β_i and the $d_{\beta_i\alpha}$ are needed.)

Its disadvantage is the number of times a node α must be subjected to equation (7). By using the storage differently and doing additional calculations, one can greatly reduce the number of nodes which are looked at, and ensure that these are looked at only once. This provides an efficient solution to the α selection problem. The algorithm A* proposed by Hart, Nilsson and Raphael[16] does not minimize $D(A,\alpha)$ directly. Instead, it maintains at each node not only the current best value of $D(A,\alpha)$, but also an estimate $h(\alpha,\gamma)$ of how far the desired terminal node γ is from α. γ may be in fact a set of nodes T, but the algorithm will tolerate the lack of any predefined γ by deciding that h is always zero. Then the computation seeks to minimize the sum

$$f(\alpha) = D(A,\alpha) + h(\alpha,\gamma) \tag{9}$$

$f(\alpha)$ is thus an estimate of the total cost of the path from A to γ which passes through α. The authors intend that values of h be determined "heuristically," using some information from the physics or logic of the problem represented in the graph.

A sensible choice for the next α is that with the smallest $f(\alpha)$. This tends to guide the process in the right direction out from A, inasmuch as f will be larger in what seems heuristically to be the wrong direction. As the work proceeds, the estimates $h(\alpha,\gamma)$ may be expected to improve in accuracy, correcting any initial tendency to move the wrong way due to initially poor accuracy in h. The authors show that a good choice for h is one which lower bounds the true remaining distance. Then only some of the nodes are looked at, and those only once.

Algorithm A* follows, with additions by this author (underlined) to handle the case in which no path exists between A and T, and to show explicitly how the paths are marked.

1. Mark A "open" and calculate f(A).
2. If the list of open nodes is empty, stop. Either there is no path or else no T was defined. Otherwise, select the open node α whose value of $f(\alpha)$ is the smallest. Resolve ties arbitrarily, but always in favor of any node α in T.
3. If α is in T, mark α "closed" and stop.
4. Otherwise, mark α "closed" and calculate $f(\alpha')$ and $D(A,\alpha')$ for each node α' adjacent to α. Mark "open" each α' not already closed and reopen any closed α' for which $f(\alpha')$ is now lower than it was when α' was closed. Indicate at each opened α' that α is the node associated with this calculation of $f(\alpha')$. (Use an arrow as in Figures 3 and 4.) Go to Step 2.

An example appears in Figure 6. Each stage except the first begins with application of Step 2 of the algorithm.

The storage and calculation requirements for A* are interrelated, since there are two kinds of nodes of interest: open, and all the rest. No matter how things are arranged, we must store, for each α, a list of the adjacent β_i, the $d_{\beta_i\alpha}$, and the "arrow" i*. After this, we have some choice. One method is to add a flag indicating "open" or "closed," and at each open node, store, in addition, f and D. This requires searching the entire graph to determine the node with

○ = Open Node ● = Closed Node

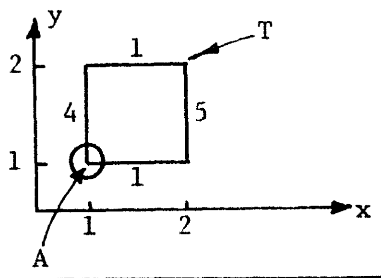
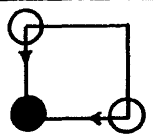
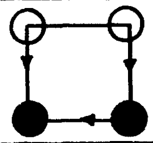
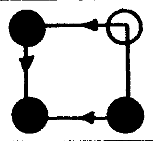
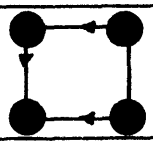
Stage	Sketch	Calculations	Nodes Opened at this Stage	Nodes Closed at this Stage
1		$f(A) = \sqrt{2}$ ($h(\alpha,\gamma)$ = Euclidean distance. One unit separates scale marks on each axis.)	(1,1) = A	
2		f(2,1) = 2 f(1,2) = 5	(2,1) (1,2)	(1,1)
3		f(2,2) = 6	(2,2)	(2,1)
4		f(2,2) = 5 (recalculated with (1,2) as a base)		(1,2)
5				(2,2)

Figure 6

EXAMPLE OF APPLICATION OF ALGORITHM A*

the smallest f. Another method is to maintain a separate open node list, containing the name or coordinate of each open node, plus f and D. This list corresponds to a sort of "wave front" of advancing calculation spreading out from A. This method requires slightly more storage but much less search time since the open node list for a cubic space of dimension M, side L and L^M nodes would contain on the order of only $(L\sqrt{2})^{M-1}$ distinct entries. Once the best node has been found, the list of adjacent nodes and the associated costs are looked up (no search required) on the graph. There are other methods. The choice depends on the capability of the computer in speed and storage.

Note that with A*, unlike the Ford algorithm or Bellman's Path Iteration method, it is unnecessary that optimal paths be found to every node. Rather, the operator may conveniently specify a terminal set T, reducing the precision with which he need specify the final state he desires, and reducing the amount of computation. However, there is always the chance that, due to obstacles or other constraints, there is no path from A to T. Then A* (as modified) loses some of its computational advantages and, like Dynamic Programming, is obliged to find paths to all the other nodes. Thus, with all the algorithms, alternate terminal nodes and paths are available to the operator without the effort of reprocessing the graph.

Multipath Solutions

A problem arises when there are many paths of equal minimum cost. This will occur very often in the grid-like graphs we customarily consider, whenever the costs are fairly uniform over the graph. (In an m x n rectangular grid graph with all costs equal to c, there are (m + n)!/(m! n!) distinct paths from one corner to the one diagonally opposite, all of cost c(m + n).) An instance of this is Example 3 of Chapter III. Here we saw that the important constraints could be expressed in a relatively simple two dimensional graph, but that some of the cheapest paths were very unsatisfactory. It would be unfortunate if our only remedy were to expand the state vector and resort to a very much larger graph, although that is a possibility. It would be better if we had some way of criticizing the paths in the simple graph, either during or after their development. Both the Ford algorithm and A* allow us to do such criticizing during processing.

Here are some of the difficulties we want to avoid: In Example 3 of Chapter III, we want paths like Figure 7a, but not like 7B:

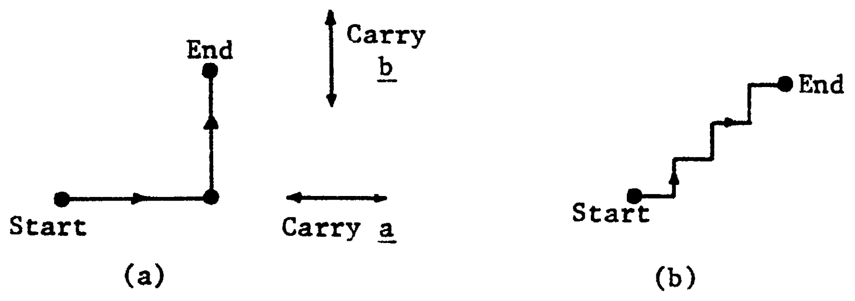


Figure 7

A GOOD PATH AND A BAD PATH

Figure 7a represents carrying a as far as it can go, then carrying b. Figure 7b represents carrying each a little way, by turns. On the other hand, with a two dimensional physical space and a two dimensional graph showing the movements of one object or the jaws, we want paths like Figure 8a, but not like 8b:

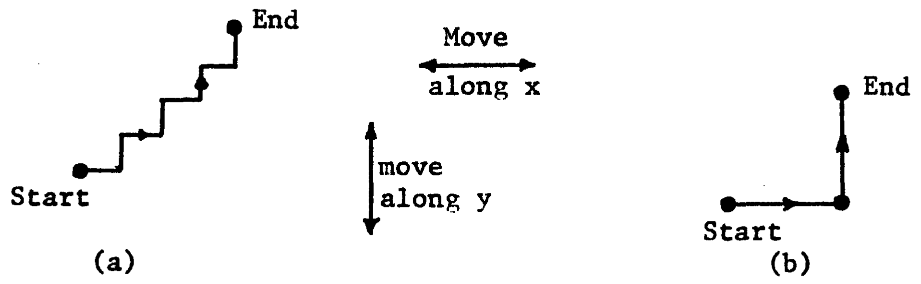


Figure 8

A GOOD PATH AND A BAD PATH

Note that adding diagonal moves in Figure 8 merely skews the paths but does not in general remove the ambiguities. See Figure 9, where a good path in the sense of Figure 7a is shown in heavy lines:

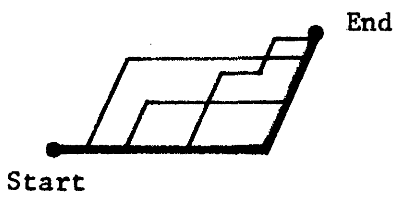


Figure 9

GOOD AND BAD PATHS WHEN DIAGONAL MOVES ARE ALLOWED

The question here is one of resolving ties. To get Figure 7a, we must resolve the ties in the same way each time, while to get 8a we must resolve them the opposite way each time. In the Ford algorithm, the opportunity comes when we determine the order in which the i's are taken in equation (8). To get something like Figure 7a, we take the i's in the same order each time, recalling that ">" is used in the definition of the algorithm. This means that ties are always resolved in favor of the first i which gives the minimum. (If "$\geqslant$" were used, the last i to satisfy the minimum wins.) If we have numbered the adjacent β_i consistently for each α (equivalently, if the commands are given subscripts in a consistent order everywhere on the graph), this guarantees that the command with the lowest subscript will get the first chance to satisfy the minimum at each α. The resulting path will contain an unbroken string of as many of this command as will be consistent with task completion and optimality. The same is true of the command with next lowest subscript, and so on. The result will be a path with few "corners." To get the best behavior out of Example 3's graph, it is wise to order the commands so that "carry a" gets subscript "1" and "carry b" gets subscript "2" at each node. To get behavior like Figure 8a, we merely try the commands in the opposite order at each node.

In algorithm A*, we achieve behavior like Figure 7a by always choosing at Step 2 that α (among those with equal minimum $f(\alpha)$) whose arrow corresponds to the command with the lowest possible subscript. Equivalently, we can use some other consistent formula which again guarantees that the same command gets the first chance to advance the wave front every time there is a tie. To get Figure 8a, we again scramble the formula.

Efficiency of Systematic and Random Procedures

Algorithm A* contains an explicit procedure for choosing the next α, while the Ford algorithm does not. We suggested selection procedures for the Ford algorithm above and will suggest another below. Is this attention to selection procedures really necessary? Why not just pick the next α for use in the Ford algorithm purely at random? We can show that this results in much less efficient computation than a procedure which selects the α's in some consistent order.

Let us define application of equation (7) to every α once as one pass over the graph. First we shall demonstrate that the $D(A,\alpha)$'s achieve their minimum or equilibrium values on any one optimal path independent of how (or if) that path is connected to the rest of the graph. In Figure 10, we show an optimal path from A to some node α_n:

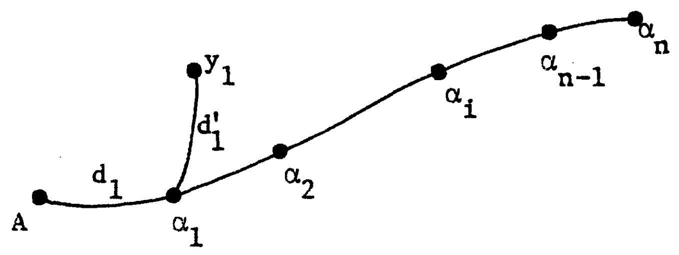


Figure 10

OPTIMAL PATH FROM A TO α_n

Let y_1 be any node adjacent to α_1 but not on this path. Then

$$D^*(A,\alpha_1) < d_1' + D^*(A,y_1)$$

because, in fact,

$$D^*(A,\alpha_1) = d_1 + D^*(A,A)$$

by virtue of the optimality of the path.

Therefore the presence or absence of the edge between y_1 and α_1 has no effect on the equilibrium value $D^*(A,\alpha_1)$. This value is determined only by d_1 and $D^*(A,A)$. $D(A,\alpha_1)$ will then take on its equilibrium value $D^*(A,\alpha_1)$ during exactly that pass on or before which $D(A,A)$ achieved equilibrium. But $D(A,A)$ is <u>initially</u> at equilibrium, so $D(A,\alpha_1)$ comes to equilibrium on the very first pass, regardless of what other nodes are connected to α_1. Applying this reasoning to α_i, it is clear that $D(A,\alpha_i)$ comes to equilibrium during exactly that pass on or before which $D(A,\alpha_{i-1})$ came to equilibrium, regardless of what other nodes may be connected to α_i.

Thus, no matter how the optimal path $\{A, \alpha_1, \alpha_2, \ldots \alpha_n\}$ is connected (if at all) to the rest of the graph, $D(A,\alpha_i)$ comes to equilibrium on or before the i^{th} pass. This means that, at the very least, one node on this path comes to equilibrium on each pass. The same is true of <u>every other</u> optimal path in the graph. As a corollary, we see that, since the longest optimal path in an N-node graph contains no more than N-1 nodes (excluding A), we need no more than N-1 passes to bring all the D's to equilibrium. Furthermore, if the longest optimal path is n nodes long (n an integer $<$ N) then processing will be complete after no more than n-1 passes. Thus we can upper bound the number of passes required by the Ford algorithm.

Now, how does pure random selection of the α's compare with this? Assume that the path in Figure 10 is the longest optimal path in a graph of N nodes. To bring this path to equilibrium, the random selector must first pick α_1, then α_2, and so on. Picking any of these α's out of order, or any other node in the graph, will not do, although it may help bring some other path to equilibrium. Thus, after $D(A,\alpha_i)$ has come to equilibrium, we await the selection of α_{i+1}. How long can we expect to wait? Since there are n nodes on this path, how long can we expect to wait until the last one has been selected in this way? This is actually a problem in Bernoulli trials, but a Markov model (Figure 11) offers a nice visualization.

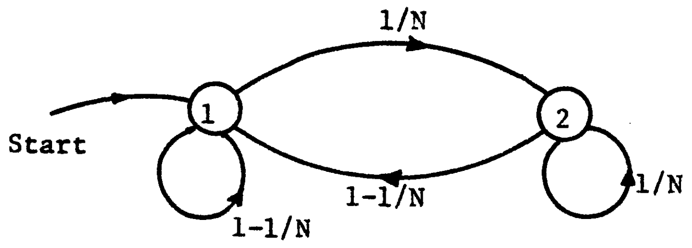


Figure 11

MARKOV MODEL FOR RANDOM STATE SELECTION

We assume that node selections are independent, and that the probability of picking any one node is 1/N. Thus node 1 of Figure 11 represents picking any node on the graph <u>except</u> the one we want. Node 2 represents picking that node. If we start in node 1 just after picking α_i, then Figure 11 says we will pick the right node α_{i+1} with probability 1/N and pick any other node with probability 1 - 1/N. In general, it will take k tries before α_{i+1} is picked, where k is a random variable. The probability mass function for k is[9]

$$p_k(k_o) = (1/N)(1 - 1/N)^{k_o - 1} \qquad k_o \geq 1$$

The mean of k is

$$\bar{k} = N$$

and its variance is

$$\sigma_k^2 = N^2 - N$$

(For large N, $\sigma_k \approx N$, indicating that almost anything could happen.) Since $\bar{k}$ is the mean number of selections required to hit each state on this path in just the right order, the mean number of selections required to process the entire longest path of n nodes is obviously $n\bar{k}$ and the variance is $n\sigma_k^2$.

Since the total number of selections is a sum of independent subtotals, we may invoke the Central Limit Theorem and say that the total number of selections needed is approximately a normally distributed discrete random variable with mean $\mu = n\bar{k}$ and variance $\sigma^2 = n\sigma_k^2$. With 97.5% confidence, we may then say that $\mu + 3\sigma$ random selections should complete the processing of the longest path. Now, the processing of every other (i.e., shorter) path in the graph has been continuing apace and independently all the while. Obviously, for such paths, the mean and variance of the number of selections required to complete processing are smaller than μ and σ^2. Thus we may have <u>greater</u> than 97.5% confidence that they too are at equilibrium after $\mu + 3\sigma$ random selections. Each confidence level, normalized by 100, is the probability that the associated path is at equilibrium. Noting the independence of the paths in this respect, we may find the total confidence that the entire graph is at equilibrium by multiplying together all these probabilities, and multiplying the result by 100. This total confidence level will undoubtedly be less than 97.5%, how much less depending on how many paths there are and their length.

If we define an <u>equivalent pass</u> over the graph as any N consecutive random selections, then we will need at the very least $(\mu + 3\sigma)/N$ equivalent passes to be reasonably sure that processing is complete. This amounts to

$$K = \frac{nN + 3n^{1/2}(N^2 - N)^{1/2}}{N}$$

equivalent passes. If $n = N - 1$, the maximum, then

$$K = \frac{N(N - 1) + 3N^{1/2}(N - 1)}{N}$$

or

$$K = N - 1 + 3\,\frac{N - 1}{N^{1/2}} \simeq N + 3N^{1/2} \quad \text{for large } N$$

or

$$K > N \quad (\text{for large } N)$$

by quite a lot. Since the Ford algorithm with systematic selection never takes more than N-1 passes, random selection is obviously less efficient.

Efficiency of Algorithmic and Heuristic Procedures

A simple Supervisory Controlled Manipulator, described in detail in a later chapter, was built, employing a plotting table manipulator and a PDP-8 digital computer.[8] The graphs typically were two-dimensional with 15 points per axis. The Ford algorithm was employed because of its simplicity. Two systematic procedures for selecting the α's were used, one moving left to right from the lower left corner, the other moving right to left from the upper right corner. Each procedure was used for two entire passes, and a tally was kept of the number of states where D was improved. As long as the paths tended up or to the right, the first procedure could be expected to select many of the α's on these paths _in path order_, thus bringing many states to equilibrium on a single pass, or at least improving their D values. The other procedure was turned to if the tally was smaller than a number called the _switching criterion_, indicating that, due to obstacles or cost structure (as in Figure 2), the paths had bent around in such a way that pursuing them from the opposite direction would be more efficient. Improvement was almost always realized by switching back and forth between the selection strategies, reducing the total number of passes needed by as much as a factor of 4 over the number needed if one selection strategy was used exclusively. Using this two-mode procedure, the graph representing the maze in Figure 12 was processed in an average of 8 passes, depending on the choice of starting state. The best switching criterion number was found empirically to be 64. Using one selection strategy all the time, the average was 27 passes.

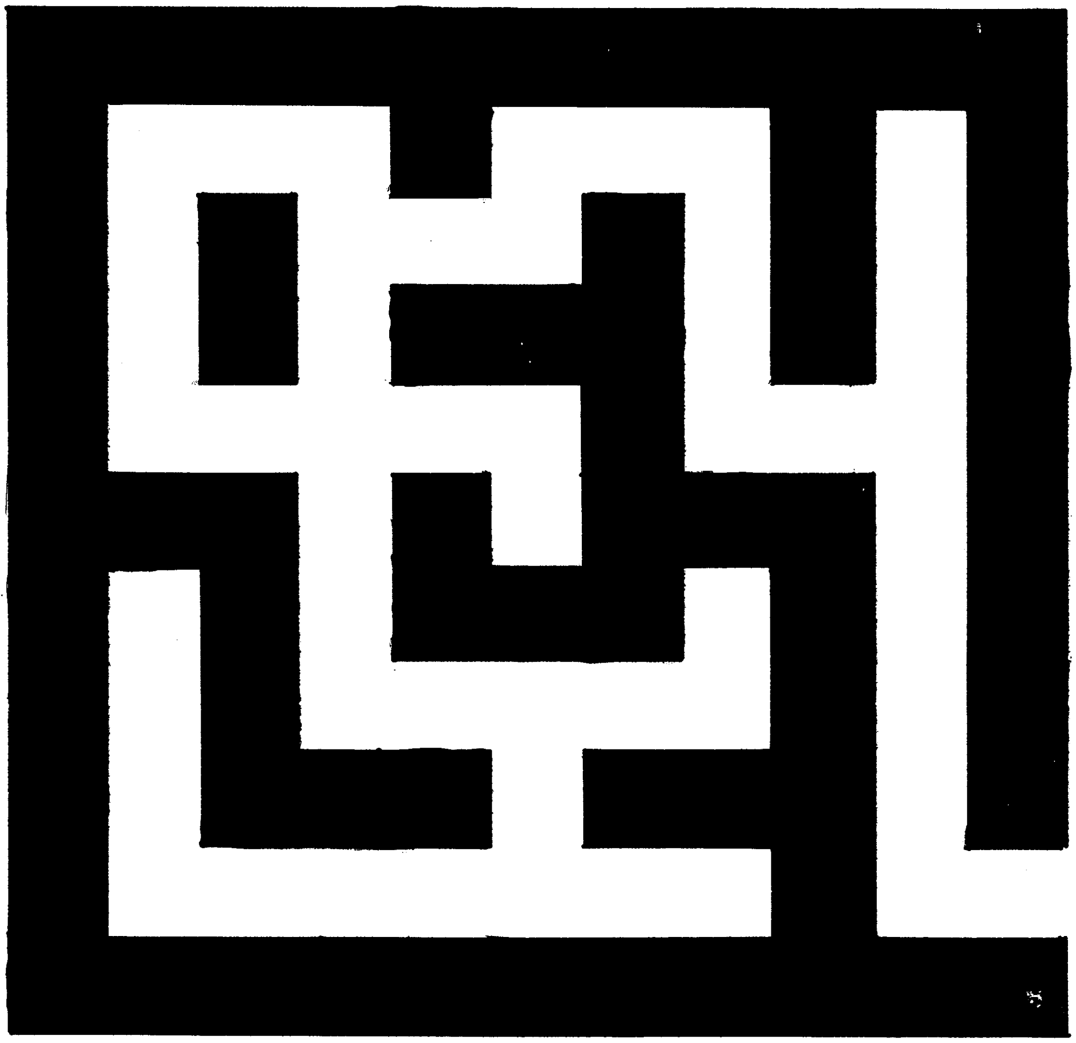


Figure 12

EXAMPLE MAZE

Computer time was about 100 milliseconds per pass, giving mean processing times of 0.8 second and 2.7 seconds respectively.

These data are presented to offer a comparison to a completely different method for finding paths across grid graphs which utilizes heuristics and will be shown to be much less efficient than the Ford algorithm. Travis[39] used IPL-V, the computer language developed by Newell and Tonge,[31] to program heuristic methods for finding paths in a 20 by 20 grid. Elementary moves were like those allowed the knight in chess. Recognizing that theorem proving and problem solving involve piecing together long sequences of elementary actions in the right order, Travis investigated a computer's ability to learn, first, that short sequences of knight's moves (the axioms) could be formed into useful motions (theorems) which made considerable progress across the grid. It was hoped that the computer could then utilize these simple theorems to prove more difficult theorems (i.e., find paths through obstacles, between widely separated points on the grid).

Travis found that "easy" problems, involving shorter spans, straighter runs, or fewer forbidden grid points, were readily solved, but some could not be solved at all. None of these "hard" problems involved as many twists and turns of the paths as does Figure 12, which, while unsuitable for knight's moves, would probably be judged "hard" by Travis' standards. We know that a graph similar to those we have been using can be drawn to show the possibilities of knight's moves on a grid. Algorithmic procedures can find existent paths without fail in a few seconds on 20 x 20 grids (although it would not have served Travis' purpose to use such methods).

Furthermore, the "hard" problems, by Travis' standards, do not take significantly longer to solve than the "easy" ones.

Because we have data on how long the Ford algorithm takes to find paths on a 15 x 15 grid graph, it is of interest to us that Travis' heuristic program took as much as 45 minutes to solve "hard" problems, and had to be stopped after an hour of effort on those it failed to solve. Even allowing that his computer was ten times slower than a PDP-8, we may therefore conclude that at least one type of heuristic method for planning elementary motions of a remote manipulator is far slower, without being more reliable, than even one of the less efficient algorithms presently available.

Taking this comparison as a benchmark, we may speculate as follows: there seems to exist a level of complexity in manipulation tasks below which algorithmic procedures can plan the motions more quickly than can heuristic methods. Chapter III's examples indicate that this level is not a trivial one in terms of the number of degrees of freedom allowed, the number of commands available, and the number of grid points to consider. Significantly higher levels of complexity in a given physical space would demand more commands and state variables, requiring of the state space method ever larger graphs. Practicality will ultimately upper-bound the complexity which this method can handle; higher level planning would then undoubtedly be taken over by heuristics or the operator. Graphs, assembled by heuristics or the operator, would be processed as before by algorithms, completing lower level planning. The saving in time and space over huge graphs hopefully would be significant. This problem is discussed in the next chapter.

Summarizing this chapter, we have developed the notion that optimal paths satisfy the Principle of Optimality and that Dynamic Programming can use this fact to find such paths. The recently developed algorithm A* was also discussed and compared to Dynamic Programming and to systematic and random methods of selecting states for calculation. Last, we compared algorithmic methods to a heuristic one and found that algorithms are faster in situations of limited complexity. We conclude with the suggestion that graphs and algorithms handle all planning below a given complexity level while heuristics and direct operator intervention handle all higher level planning.

CHAPTER V

EXTENDING THE POWER OF THE STATE SPACE MANIPULATION METHOD

In Chapter II we saw that a state space graph is a map of all possible combinations of all the commands in a limited, static set. The examples of Chapter III showed that many basic manipulations could be modelled with such graphs. In Chapter IV we demonstrated that suitable search methods could cull from a graph that particular combination of commands which would perform a stated task. Environments in which there are many objects and many allowed commands will require, with workable quantization levels, impractically large graphs. This chapter discusses ways in which realistic manipulation tasks can be planned without recourse to such huge state spaces. The method is to involve the human operator in establishing subgoals, thereby reducing greatly the dimensionality of the problem without degrading the quality of the solution.

Large graphs are undesirable not only because of their size. The purpose of a large state vector is to express all the various combinations of objects' and jaws' positions so that all of the "possible" moves may be evaluated. The trouble is that the majority of such moves are <u>not</u> possible. For example, unless the wind blows, object 3 and object 7 cannot move together while the jaws and the other objects remain fixed! The result is that the many points, each representing a new arrangement of objects and jaws, are connected by only a very few lines, indicating that only a tiny fraction of the moves dutifully investigated by the search algorithm can occur. In spite of the fact that there is a certain beauty in the high dimension solution of complex manipulation problems, the space and time consumed are not being utilized fully and in any case are both larger than practical limits.

A consequence of the sparseness of lines in such graphs is that, regardless of their dimension, these graphs yield up solution paths with a quite simple property: the paths are piece-wise planar (for a two-dimensional work space) or piece-wise prismatic (for a three-dimensional work space). That is, they lie on connected sequences of plane or prismatic cross-sections through the graph.* The result is that the solutions look

---

* For an N dimensional graph on a two dimensional physical space, let the state vector be $X = (x_1, x_2, \ldots x_N)$. Suppose (x_1, x_2) describes the jaws' coordinates and succeeding pairs describe each object's location. (Orientation is not considered.) When the jaws move empty, all x_i for $i > 2$ are fixed, so that the corresponding path lies on that $x_1 x_2$ plane which is determined by the constraint that the other N-2 variables (the objects' locations) remain fixed. If we grasp and move the object whose location is described by (x_k, x_{k+1}), then $x_k = x_1$ and $x_{k+1} = x_2$ all along the corresponding path. This path therefore lies on that plane determined by the other N-4 fixed variables plus the two constraint equations

$$\begin{aligned} x_k &= x_1 \\ x_{k+1} &= x_2 \end{aligned} \qquad (1)$$

(footnote continued on next page)

suspiciously alike in some way, although they differ in detail. A pattern can be abstracted from them which reads:

.... move jaws to a thing, grasp it, carry it somewhere,
(do something with it and the environment, carry it somewhere) release it...

The part in parentheses applies when the "thing" is a tool. Such a sequence can serve as a basic manipulation operation. Call it the "Pick-Put-Do-Put" unit. If the "thing" is an object being moved, the part in parentheses is ignored, and we get the "Pick-Put" unit.

For example, we could use "Pick-Put" to place a screw on the end of a screwdriver, and then use "Pick-Put-Do-Put" to carry the two to the threaded hole, spin the screwdriver clockwise until the torque level reaches some criterion, then return the screwdriver

*(continued) making a total of N-2 constraints or two degrees of freedom. If the jaws grasp a pusher object and push a second object (x_j, x_{j+1}), then we have N-6 fixed variables plus

$$\begin{aligned} x_k &= x_1 \\ x_{k+1} &= x_2 \\ x_j &= f_1(x_1) \\ x_{j+1} &= f_2(x_2) \end{aligned} \qquad (2)$$

where f_1 and f_2 are fixed relations which indicate that the pushed object is adjacent to the pushing object and the jaws during pushing. This again gives N-2 constraints or two degrees of freedom so that again the path will lie on some plane within the graph.

That solution paths should lie on plane cross-sections of N dimensional graphs is quite a restriction. For example, a general N dimensional graph could support paths in which, at a given transition, all N state variables could change. In a manipulation state space, this would correspond to a case of mass St. Vitus' Dance among the objects.

to the workbench or toolbox. "Pick-Put" units are good for assembling objects into larger objects, while "Pick-Put-Do-Put" units are good for applying specialized actions like force or torque. Both require various types of feedback for their execution.

The fact that such an abstraction is possible shows that solution paths through large manipulation graphs are not themselves very complex. What this means is that although the problem posed may be N dimensional, the solution is a sequence of two- or three-dimensional strokes. The significant parts of the paths are the junction points between the strokes. For simple "Pick-Put":

```
.
.
.
move empty until at $Z_1$
grasp                                          (3)
move full until at $Z_2$
release
.
.
```

Z_1 and Z_2 could be thought of as subgoals or equivalently as the junction points between the two- or three-dimensional strokes "move empty" and "move full." Many of the complex tasks which could be carried out on an N dimensional graph will have solutions made of chains of elements like (3). The beauty of the formal solutions is that the points Z_1, Z_2, ... are chosen automatically and optimally without intervention by the operator. The disadvantages have been cited above. Thus we should seek some way of using the formal similarities between solutions by involving the operator in the selection of the subgoals. The foregoing should make it clear that two- or three-dimensional graphs can form a basis for such procedures.

A Minimal State Vector

The most likely state vector for such low-dimension graphs, consistent with the "Pick-Put" idea, is $(x, y, z)_{\text{Jaws}}$. Certainly, these variables cannot be ignored completely. If we concentrate on them exclusively, moreover, we can plan most any manipulation task. This cannot be said of the variables describing the location of some object which is not involved in whatever manipulation is currently going on. Examples 1, 3, and 4 of Chapter III yield only a partial solution to the problems they posed; the required jaw motions, especially as they are influenced by the locations of the objects, have yet to be evolved. The solutions produced by these examples are notable because they comprise one or more "Pick-Put" units in the correct order and with all locations specified. In more complex situations, such as Example 6 with several objects blocking the door, there may be no sufficiently simple graph which will deliver the necessary "Pick-Put" units. Then the operator may have to supply them. In any case, a method of converting long strings of "Pick-Put" units into the required detailed jaw motions would be very helpful.

To accomplish this, the locations of objects must be known, to be sure, both so that the jaws can move without collisions and so that the consequences of moving the objects may be kept track of.

Yet these locations need not be given the status of state variables in order that manipulation on an elementary level be possible. What can and cannot be done at this level will become clear in what follows.

It is obvious that, with this elementary state vector, more may be required of the operator, since the available commands consist only of instructions to move the jaws around. These commands correspond to the transitions whose availability at various points in the physical space is kept track of on the graph. Even grasping and releasing are not immediately available; they must be handled separately.

But this is not so difficult. The approach consists of making more use of the information already available in the two- or three-dimensional graph of jaw motions. In such a structure, objects appear merely as forbidden areas. Thus the only benefit to fall out of this graph automatically is that paths deduced for the jaws to follow will avoid all known objects. This benefit is the first step up from pure manual control, and is of great utility when there is delay or when the operator cannot see all of the objects. However, this benefit prevents grasping since, ironically, every object is off limits to the jaws. An easy solution exists, however, called command perturbation. To use it, we need only append to the graph a directory of object names and their physical locations. Consider this situation: Object A lies on a table. A pair of non-rotating jaws can move about on the table. See Figure 1.

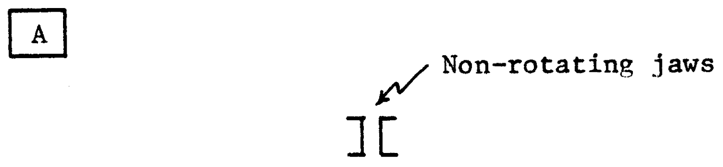


Figure 1

JAWS AND ONE OBJECT

Object A is to be grasped. The graph of allowed jaw motions near A looks like so:

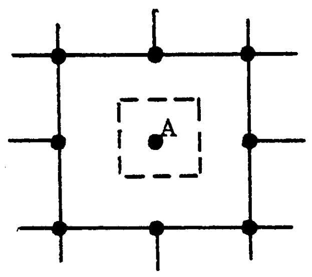


Figure 2

GRAPH OF JAW MOTIONS NEAR OBJECT A

Now if we blithely reconnect A's location to the graph like this:

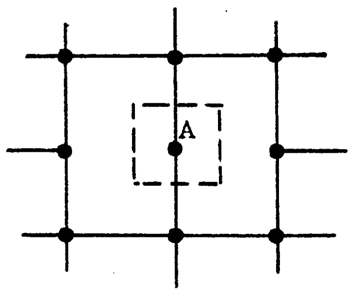


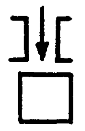
Figure 3

MODIFICATION OF THE GRAPH NEAR A

and ask for a path which will take the jaws to A's location, then the resulting string of commands will terminate

$$\ldots u_{n-1}, u_n.$$

where u_n is either the move shown in Figure 4a or 4b:


(a)

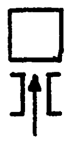

(b)

Figure 4

WAYS OF GRASPING

whichever move yields the cheapest path. However, no good comes of this unless the jaws are open prior to u_n, for otherwise they will merely crash into the object. A solution is to modify the command string to read

$$\ldots u_{n-1}, \text{open}, u_n, \text{close}.$$

Then this string, when executed, will result in the object being grasped by the jaws, regardless of the initial position of either. Thus the command "Grasp named object" is easily added to the repertoire without adding any dimensions to the graph.

With an object in the jaws, the command "Carry to a named location and release" is obtained in a similar way: Delete from the named location on the graph of jaw motions all approaches except the two along the grasp-release axis:

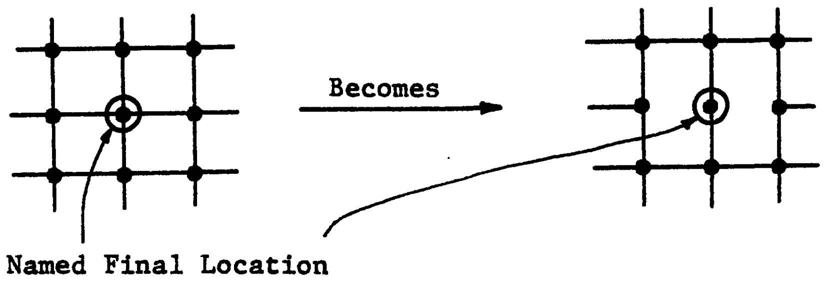


Figure 5

MODIFICATION OF GRAPH NEAR RELEASE POINT

A path for the jaws to follow to this location would end with the commands

$$\ldots\ u_{n-1},\ u_n\ .$$

where u_n is a move along the grasp-release axis in the optimal direction, if such a move is possible given the constraints in the neighborhood of this location. Since we know that the last direction is clear if this path exists, we may be sure that the following modification will also exist:

$$\ldots\ u_{n-1},\ u_n,\ \text{open},\ -u_n,\ \text{close}.$$

where $-u_n$ signifies moving in the direction 180° opposite to that called for by u_n. The resulting path causes the object to be left at the named location and the jaws in an adjacent location, closed.

So, with no increase in dimensionality, that is, without making the location of the object a state variable, we can add grasping and carrying of named objects to our ability to avoid obstacles. Basically, the trick is to consider every object an obstacle unless we designate it as the one we want to move. This partition of the environment into one "object" and the rest "obstacles" makes sense and is consistent with what people probably do. By contrast, the inefficiency of multi-dimensional structures is exactly that they insist on considering the motion of all the objects in all combinations, regardless of the situation. Thus the operator, by naming his "object" reduces the dimension of the problem space. The solution is no less optimal.

A step up from this level allows us to name the object and its terminal location all at once and have the path generated automatically. There exist two interesting approaches. In the first, called chain graphs, we use a sequence of two- or three-dimensional graphs tied together with special directed arcs. Say we have the situation shown in the graph of Figure 6. We wish to take object A to location X. The jaws must move empty to A, avoiding

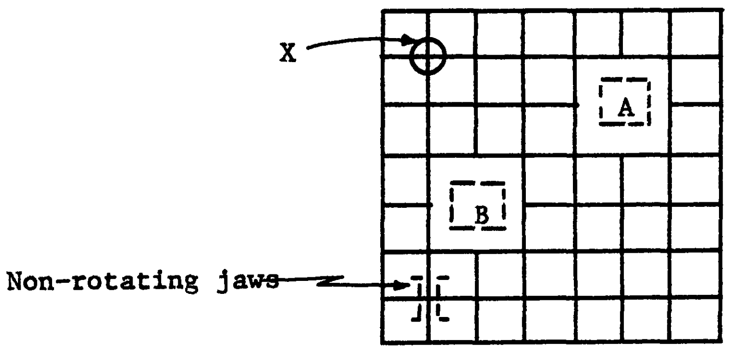
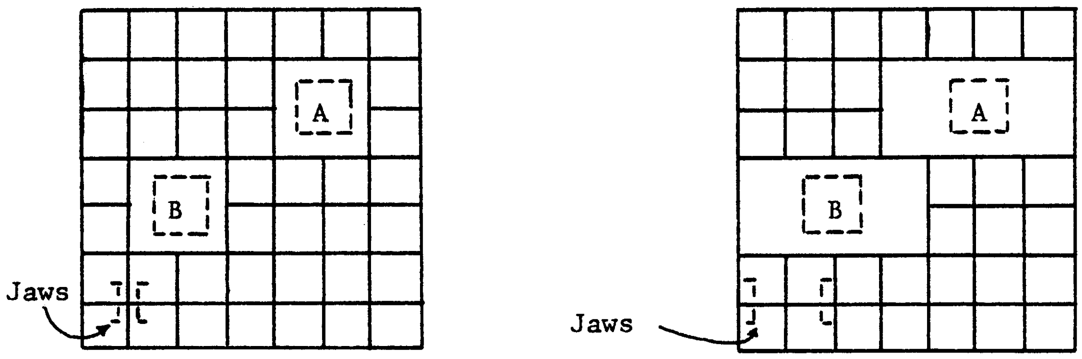


Figure 6

EXAMPLE GRAPH WITH OBJECTS INDICATED BY MISSING EDGES

B, grasp, move full to X, release and back off. The graph on which moves for full jaws are planned is different from that on which moves for empty jaws are planned. (The same is true in planning "grasp" and "carry" above.) The reason is that full jaws are bigger than empty closed jaws. Of course, from the jaws' point of view, it is the objects which are bigger. So it is easy to indicate a full or wide open jaws situation on a graph of allowed jaw motions by making the objects bigger in the directions in which the jaws are extended when open. We do this by deleting lines from the graph. Let us put the Empty Jaws graph next to the Full Jaws graph like so:



Jaws Closed Empty
(a)

Jaws Full or Open
(b)

Figure 7

EXAMPLE GRAPHS SHOWING STATE OF JAWS

Now if we connect the grasp approaches to A's initial position on the empty graph to its initial position on the full graph using a directed arc called Grasp, we have Figure 8:

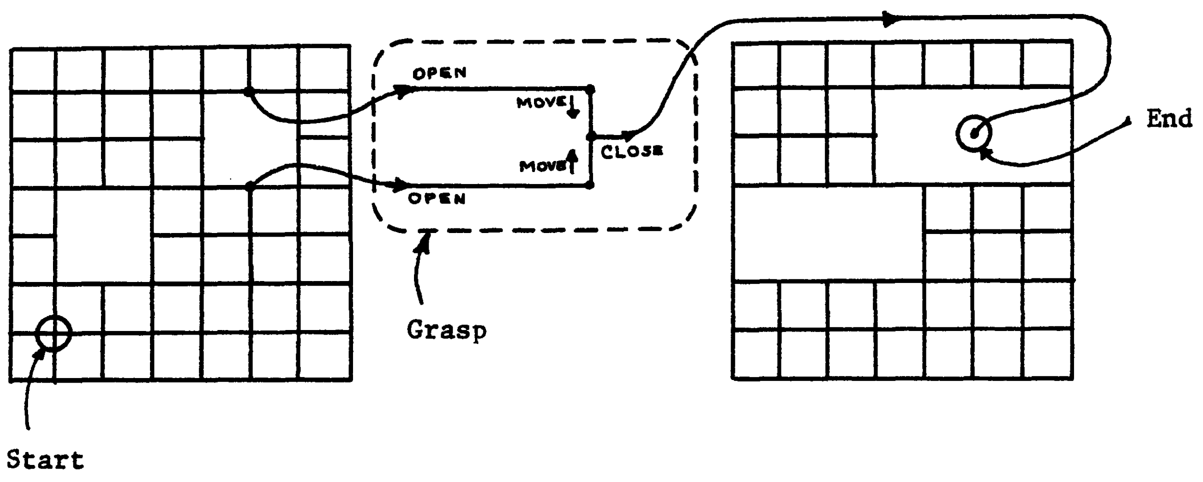


Figure 8

GRAPH WHICH ALLOWS "GRASP A" TO BE DEDUCED DIRECTLY

If we ask for a path on this pair of graphs starting at the point labelled "start" and ending at the point labelled "end," we obtain "Grasp A." If we do this again with the desired terminal location for A, however, we can get grasp, carry and release all in one bite. See Figure 9. (On the Full graph in the center, we have connected A's location into the graph completely, since, once A is grasped, restrictions on approaching it do not apply.) In fact, a completely arbitrary end point for the empty jaws may be specified. The approach directions in grasp and release are chosen optimally with respect to where the jaws came from before grasping and where they are to go after releasing. Here we use three two-dimensional graphs simultaneously. The grasp and release links are added especially for the evaluation of the given command, which must specify A, its new location, and the jaws' new location. The solution path is exactly the same as the one we would have obtained from a formal

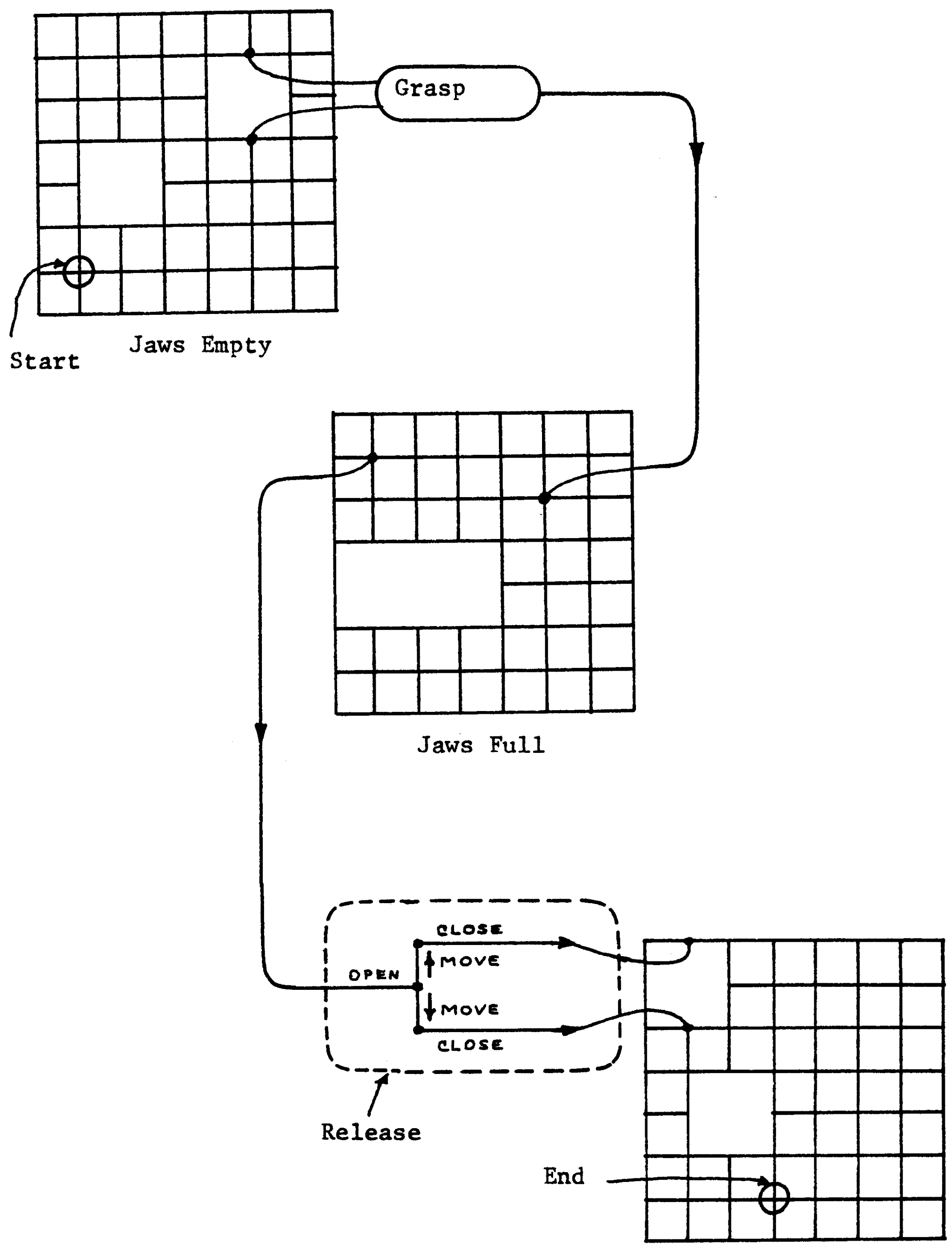


Figure 9

GRAPH WHICH ALLOWS "TAKE A FROM X TO Y"

solution on a graph of 5 dimensions ($(x,y)_A$, $(x,y)_{Jaws}$, H) having 2×8^4 points (2×8^6 points if we include $(x,y)_B$). By contrast, Figure 9 has 3×8^2 points, a reduction of a factor of 40 or more in both storage space and computing time. With realistic quantization and a three-dimensional work space, the reduction would be vastly greater.

It should be clear from this that we do not need the 5-dimensional graph. Once we give the command "Take A to X and leave the jaws at Y," we have specified three key points along the path. Only the parts of the path between these points need be found. These parts lie wholly on simple two-dimensional cross sections whose coordinates are completely specified by the given points. The formal solution ignores these points, except as they specify the _end_ of the path in the bigger space. The chain graph method basically extracts these cross-sections and employs them directly.

However, this solution method cannot be extended too far due to space limitations. If we wish to plan out in advance a considerable sequence of moves, we would run up quite a string of graphs whose _simultaneous_ reduction by a search algorithm is not really necessary. In fact, simultaneous reduction of the three graphs in Figure 9 is not really necessary, either. The human operator must specify two points on each graph. All the algorithm must do is find a path which hits those points. This can be done easily by processing the graphs in sequences of pairs, and then sticking the resulting substrings of commands together to form the final solution path. The first substring is deduced by applying the algorithm to the left and center graphs of Figure 9, using the jaws' initial position on the left graph for "start," and A's final location on the center graph for "end." The second substring is deduced from the center and right graphs of Figure 9, with A's final location on the center graph for "start" and the jaws' final location on the right graph for "end." Approach directions for grasp and release are still chosen optimally with respect to the entire task. This pairwise utilization of low dimension graphs can be carried on indefinitely using only the space required for one pair, plus the much smaller space needed to store the growing path. Each pair of graphs is linked with a grasp or release connection, with the requisite information supplied by the operator as far in advance as he wants or dares.

Manipulation Functions

This approach may be used as the basis for giving the operator the ability to define manipulation functions. For example, the simple function Take(A, X, Y) is defined by the operator in real time (in some appropriate interpretive computer language) as:

$$\text{Take}(A, X, Y) = \text{Pick Up}(A) + \text{Carry}(X) + \text{Move}(Y) \qquad (4)$$

The functions Pick Up, Carry, and Move may be defined internally by a program which generates grasp and release connections between pairs of graphs as in Figure 9. Location functions such as Next To (B), Top of (C), and so on, can also be defined and <u>used as arguments</u> for Pick Up, Carry and Move. The function Take, once defined, can be used again and again with any arguments. The program builds the required graphs and finds the path. This saves space over permanent storage of a big graph containing in effect all possible grasp and release links, and is more convenient for the operator than issuing Pick Up... individually each time. In the latter case, approach directions for grasp and release would not be chosen optimally with respect to the entire task. A more general way to define manipulation functions will be discussed below. Suffice it to say here that function definition is a basis for a computer manipulation language extension of graph methods. (Cf. Barber's MANTRAN, op. cit.)

We can use the above methods to solve completely the spar problem discussed in Example 4 of Chapter III. We begin with the spar and jaws each located and oriented arbitrarily. We specify terminal locations and orientations for each. However, instead of using a 6-dimensional graph, we shall use 3 three-dimensional graphs, strung together with grasp and release arcs especially for the execution of this task. The state vector $(x, y, \alpha)_{\text{Jaws}}$ suffices*, since everything can be expressed in terms of the orientation and location of the <u>jaws</u>, provided we distinguish empty jaws from full.

The empty jaws can go through any doorway in either orientation, but the full jaws (holding the object in the middle) are restricted in their motions to exactly those illustrated by the graph in Figure 8, Chapter III. Suppose the jaws are initially in

---

* Orientation is expressed by α, such that orientation parallel to the x axis corresponds to $\alpha = 0$; parallel to y corresponds to $\alpha = 1$.

orientation $\alpha = 0$ and we want them to end up with $\alpha = 1$. The rest of the task is the same as in Chapter III. During the task, the jaws must rotate empty to grasp the spar, rotate while carrying as many times as necessary, and rotate once more after releasing. The spar's initial location and orientation contain all the information required to place the grasp arc between the correct points on the correct graphs. The operator's stated final locations of spar and jaws tell everything needed to place the release link and specify the final state. In both grasp and release, approach or retreat directions may be inhibited by constraints of the environment.

The result is sketched in Figure 10. Each plane has coordinates for x and y location of the jaws. The center pair appear separately as Figure 8 of Chapter III. The pairs to the left and right are similar in appearance, except that they contain fewer missing links, reflecting the greater freedom of movement of the empty jaws.

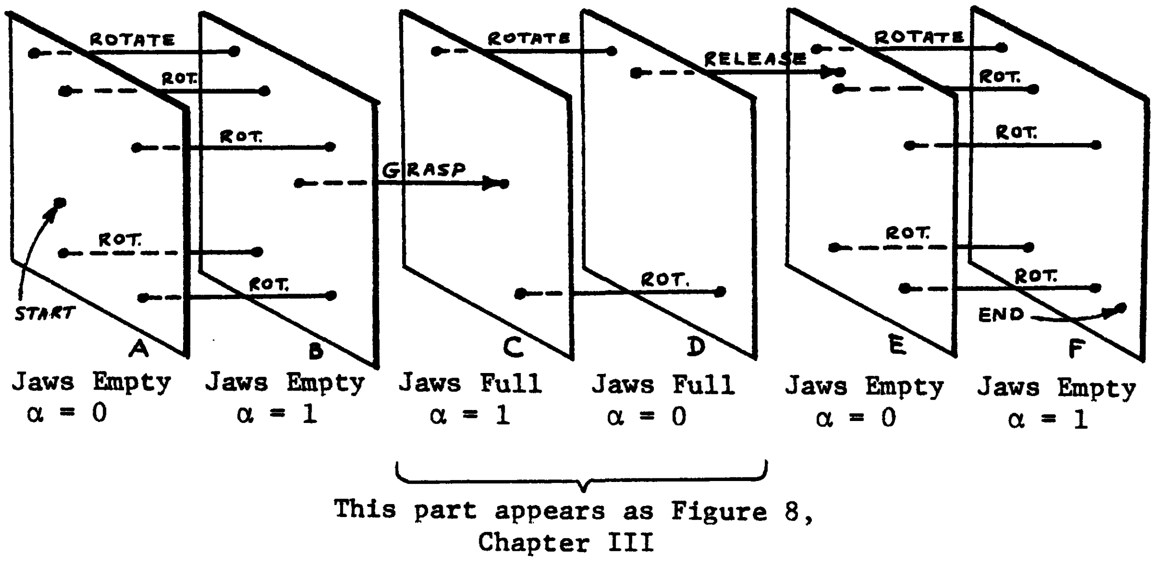


Figure 10

CHAIN OF GRAPHS FOR THE SPAR PROBLEM

Note that there are many Rotate links between graphs A and B, and between E and F. These are part of the graphs and not supplied by a special program. Their large number indicates that Rotate can occur at many places in physical space as long as the jaws are empty. By contrast, there are few Rotates between graphs C and D, reflecting the restrictions which the environment places on the full jaws. Were the spar's final orientation specified as $\alpha = 1$, then a release link would be drawn from graph C to graph F instead of D to E, as in Figure 10. The appropriate solution emerges in either case.

A graph built to order, such as Figure 10, is good only for the stated task. But this lack of generality is its great advantage, because it concentrates on the desired task rather than on <u>all</u> tasks. It is therefore vastly smaller and still does the same job as would the general 6-dimensional graph.

The previous discussion brings out one main point -- the graph approach to planning manipulation tasks may be extended in power and versatility by means other than increased dimension of the graph. Instead, we make a basic shift in attitude, and say that graphs of a certain maximum size are themselves to be used as the elements of a larger structure. Graphs of limited dimension allow us to define directly such actions as Move, Rotate, Open and Close, and indirectly, Grasp, Release, Pick Up and Carry. Then Pick Up and Carry are used to define more complex actions like Take. Then Take may be used to define Switch, for example:

$$\text{Switch}(A,B) = \text{Take}(A,X,Y) + \text{Take}(B,Z,Y) + \text{Take}(A,U,Y) \quad (5)$$

where X is a location between A and B, Z is A's old location, U is B's old location, and Y is some convenient stopping place for the jaws. The result is that A and B trade locations. Each step is the foundation for the next step; each molecule is the atom of the next higher level of complexity.

The cement which holds these chain graphs together is language -- the commands issued by the operator. Thus our shift in emphasis can be identified as linguistic. When we allow the operator to say "Take A to X and leave the jaws in Y" without requiring a 5-dimensional graph, we are making better use of what he says.

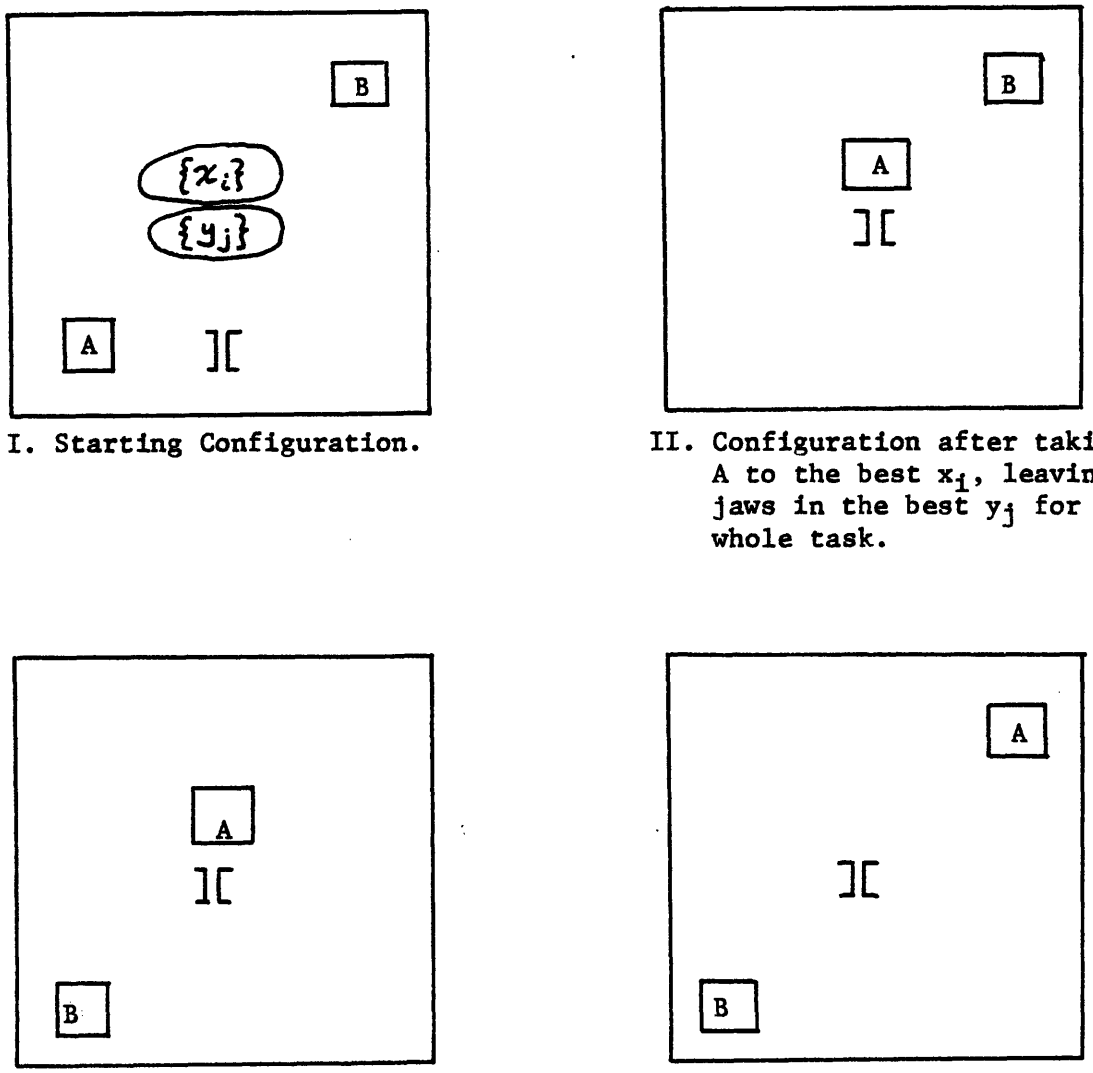
More Complicated Manipulation Functions

We may go beyond simple functions like Take and Switch. For example, the function Switch(A,B) may be "optimized" if we have a way of inputting this kind of definition:

$$\text{Switch}(A,B) = \min_{\substack{\{x_i\} \\ \{y_j\}}} \left[\begin{array}{c} \text{Take}(A,\ x_i,\ y_j) + \text{Take}(B,\ Z,\ y_j) \\ + \text{Take}(A,\ U,\ y_j) \end{array} \right] \tag{6}$$

Here, $\{x_i\}$ is a set of locations specified by the operator, lying between A and B, of which the optimal one is to be chosen automatically when the function is evaluated. $\{y_j\}$ is a set of locations similarly given by the operator at which to leave the jaws, of which again the best is to be chosen automatically. In this definition of Switch(A,B), only one y_j will be chosen. The result might be as depicted in Figure 11.

The evaluation of equation (6) requires the establishment of a tree representing the result of choosing each x_i and y_j. The cost of each arc on the tree is obtained by constructing the graph chain which represents the action called for by this arc, say



I. Starting Configuration.

II. Configuration after taking A to the best x_i, leaving jaws in the best y_j for the whole task.

III. After taking B to A's old location, leaving jaws in same y_j as in step II.

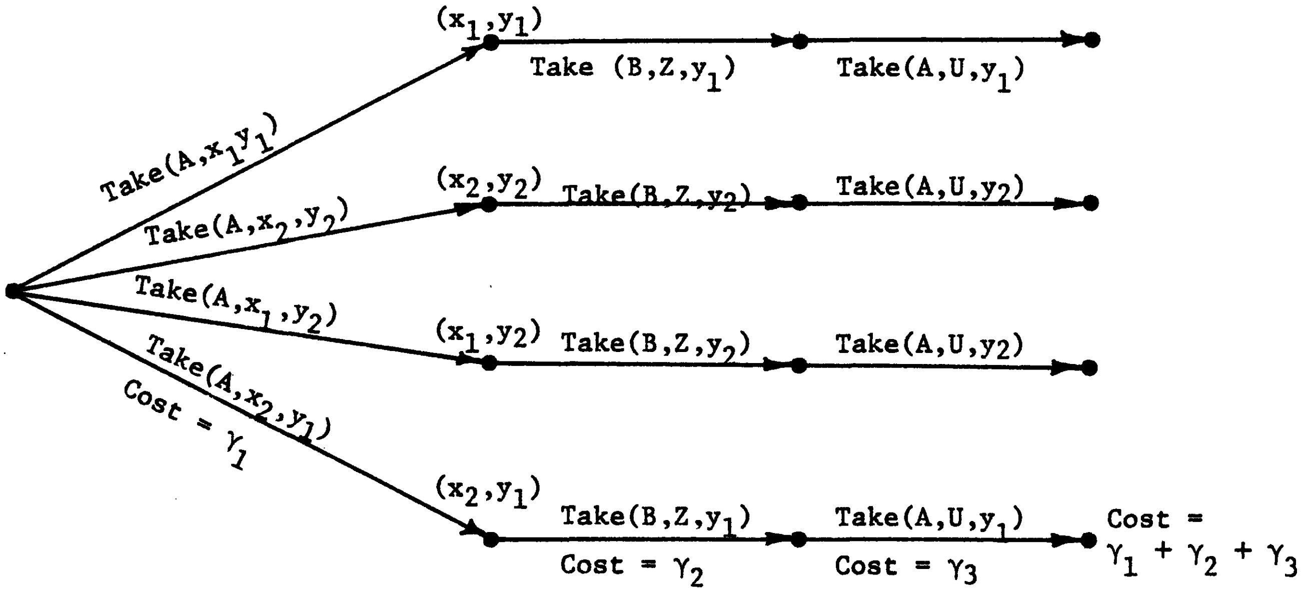
IV. After taking A to B's old location, leaving jaws at same y_j as in steps II and III.

Figure 11

FOUR STAGES OF PLAN FOR SWITCH(A,B) WITH OPTIMAL STOPPING PLACES FOR A AND JAWS DEDUCED AUTOMATICALLY

Take(A, x_3, y_7). The best way of doing this command and its cost are then deduced from the graph chain by our usual methods. When all the possibilities in $\{x_i\}$ and $\{y_j\}$ have been priced, the cost of Switch for each choice of x_i and y_j will emerge. From this, the cheapest x_i and y_j may be chosen, completing the evaluation.

For example, let $\{x_i\}$ have elements x_1 and x_2, and let $\{y_j\}$ have elements y_1 and y_2. Then the tree is



Note that, in spite of their <u>name</u> similarity, duplicate occurrances of Take(B, Z, y_1), for example, need not have the same cost. The reason is that part of Take's cost is in moving the jaws empty from their initial location to the object. In each case, the initial location is different, so the cost will in general be different. (In this example, this is not true of duplicate occurrances of Take(A, U, y_1) or of Take(A, U, y_2), since in the former case, the jaws always start from y_1, in the latter always from y_2.)

We could have defined Switch as

$$\text{Switch}(A,B) = \min_{\substack{\{x_i\}\\ \{y_j\}\\ \{v_k\}\\ \{w_\ell\}}} \left[\begin{array}{c} \text{Take}(A,\ x_i,\ y_j) + \text{Take}(B,\ Z,\ v_k) \\ + \text{Take}(A,\ U,\ w_\ell) \end{array} \right] \qquad (7)$$

allowing different stopping places for the jaws at each stage. This would complicate the tree a great deal and would not improve the solution much if the operator were sensible in his choice for $\{x_i\}$ and $\{y_j\}$ when using (6).

Equation (6) renders possible a good approximation to the formal optimal solution produced by a 7-dimensional graph $((x,y)_A$, $(x,y)_B$, $(x,y)_{Jaws}$, $H)$. We have vastly reduced the calculation by allowing the operator to designate $\{x_i\}$ and $\{y_j\}$. Were he to let these sets be as big as the entire space, (6) would have to evaluate all the object motion combinations which could possibly satisfy the task. Yet even this would involve less calculation than the formal solution, since the latter would also investigate all the other move combinations which could <u>not</u> possibly satisfy the task, and which vastly outnumber those which could. Involving the operator in this way can therefore effect huge reductions in computing load. The resulting function is, like the simpler one in (5), available for use in the future with any objects (arguments) A and B. The operator need not define it each time he wants to use it.

Recursive Manipulation Functions

More interesting functions than Switch can be defined to perform manipulation. Recall Example 6 of Chapter III, the blocked doorway problem. Here we built the state space so that the computer could automatically clear the doorway of the blocking objects, picking just enough of them in just the right order and moving them just far enough out of the way. This is analogous, in a crude way, to the situation in which subtasks must be conceived and executed before the stated main task can be accomplished. If four or five objects block the door, there results a very complex sequential decision problem. Since an operator with even limited vision can solve such a problem almost effortlessly, it is more efficient, if less elegant, for him to help the computer out.

In fact, not much help is needed to reduce the problem from one of 2(N+2) dimensions (for a 2-dimensional work space, N blocking objects, one object to be moved through the door, and jaws) to a handfull of problems of two dimensions each. An example follows.

Let us consider first, as before, one blocker called B. See Figure 12.

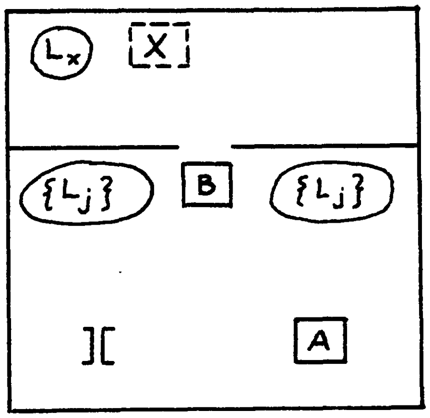


Figure 12

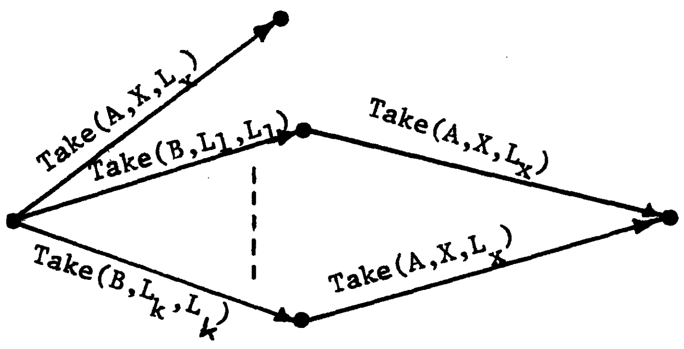
ONE OBJECT BLOCKS THE DOOR

The disjoint set of locations $L = \{L_j\}$ is a region of the physical space designated by the operator as "out of A's way." A location for B is to be chosen optimally from this set. L_x is the final jaw location. A function which will handle this is

Move B out of the way and/or move A to X =

$$\min \left\{ \text{Take}(A, X, L_x), \ \min_{\{L_j\}} \begin{bmatrix} \text{Take}(B, L_j, L_j) \\ + \text{Take}(A, X, L_x) \end{bmatrix} \right\} \qquad (8)$$

"And/or" is simply a convenient way of stating that A should be moved directly to X without moving B if that would be cheaper. (The use of L_j for the jaws' final location after moving B makes for great simplification.) The tree for evaluating this looks like so:



The operator, by specifying the set L, reduces the computational load tremendously. He is designating a region in which subgoals should be sought. This is easier for him than choosing the subgoals with precision and easier for the computer than no choice at all. Optimality is obtained within this choice range.

Our objective is to generalize (8) to handle any number of blocking objects, and in particular to be able to move just enough of them to make optimal motion of A possible. We need the following definitions:

For convenience, rename $L_x = L_o$ and $A = B_o$

L	=	$\{L_o, L_1, \ldots\}$ = a set of locations: L_o = A's final location; $L_1, L_2, \ldots$ = possible new locations for blockers.
B	=	$\{B_o, B_1, \ldots\}$ = a set of objects: B_o = A; $B_1, B_2, \ldots$ = blockers.
M	=	an ordered set of objects belonging to B, which have already been moved to locations in L.
F	=	an ordered set of locations in L to which objects have already been moved.
B - M	=	objects not yet moved.
L - F	=	locations not yet filled.
ϕ	=	any set with no members (the empty set).
G(M,F)	=	the least cost of having moved objects M, in order, to locations F, in order.

Then

$$G(M,F) = \min_{\substack{B_i \varepsilon B-M \\ L_j \varepsilon L-F}} \left[\begin{array}{l} \text{return jaws and} \\ \text{for } i = 0,\ \text{Take}(B_o,\ X,\ L_x) \\ \text{for } i > 0,\ \text{Take}(B_i,\ L_j,\ L_j) \end{array} + G(M-B_i,\ F-L_j) \right] \qquad (9)$$

($B_i \varepsilon B-M$ = any B_i not yet moved). A branch is to be terminated as soon as B_o has been moved, regardless of which other B's have been moved. If L is exhausted before the doorway is cleared, the problem is insoluble as stated. At the beginning, no objects have been moved and no locations have been filled. Thus $M = F = \phi$, and we have the boundary condition

$$G(\phi,\phi) = 0 \qquad (10)$$

In case there are no blockers, $B = B_o$ immediately and the tree reduces to a single branch. An example appears below.

The important thing about (9) is that it is recursive: the procedure for choosing the next blocker and where to move it is the same regardless of how many blockers are left. This function is therefore defined for any number of blockers (including none), as long as there is enough space in L to hold the minimum number of blockers which must be moved to allow A's passage. (This means that the operator, by specifying a larger L, could perhaps get the task done more cheaply, but with more computation.) M and F are ordered sets because the order in which the B_i and L_j are chosen affects the cost.

Then equation (9) will deduce the optimum order in which to select just enough of the blockers and the optimum location to take each to, while maintaining minimum cost for the given L. The operator need not know which of the B_i will be moved, but may designate for consideration as many as he thinks necessary. The formal solution, using a 9-dimensional graph, also selects the minimum number of blockers needed to achieve minimum total cost. However, equation (9) uses the operator two ways to reduce the dimension of the problem and the extent of each dimension: First, he selects L intelligently, while the formal solution must use all unoccupied space for L. Second, he may be able to see that some of the B_i will not block any acceptable path for A, and can just leave them out of set B. The formal solution must consider them all. Equation (9) corresponds roughly to the command "Move enough of those things there out of the way so that you can take A to X." This is definite only where it needs to be, concerning A and X. The rest is suitably vague and the computer can generate a solution to its liking or report that no solution exists.

In Figure 13, we show an example with three elements in B and four in L:

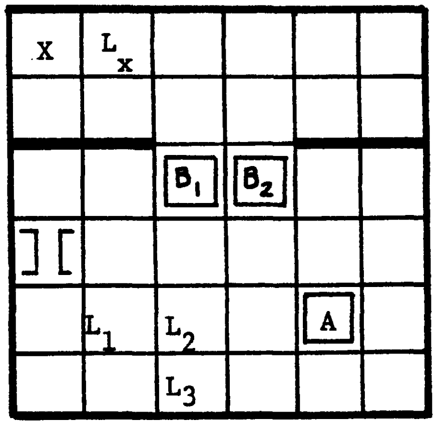


Figure 13

MORE COMPLICATED BLOCKED DOORWAY PROBLEM

The cost of moving empty one unit is one (1), and the cost of carrying one unit is two (2). No diagonal moves are allowed. In

Figure 14 we show the resulting tree. Path costs are shown, along with two examples of G(M, F). Path cost of ∞ implies no path exists. The cheapest path across the tree indicates the following solution, expressed in "Pick-Put" units:

Put B_1 in L_3

Put B_2 in L_2

Put A in X

The solution requires 37 "Pick-Put" calculations on 2-dimensional trios of graphs like Figure 9. Each trio has 108 points, each point has at most 4 neighbors. It takes about 90 micro-seconds of PDP-8 time to apply the Ford algorithm to one pair of points. If 107 passes are required, the maximum, then it will take about

$$9 \times 10^{-5} \times 4 \times 108 \times 107 \times 37 \cong 147 \text{ seconds}$$

to compute the optimum path.

The formal solution requires reduction of two 8-dimensional graphs of 6 points per axis (H is quantized to two points), or 2×6^8 points, each with at most 2 x 8 or 16 neighbors. If a point is connected to half its neighbors on the average, and if index equilibrium is reached in only 10% of the maximum number of passes, then the computing time is about

$$9 \times 10^{-5} \times 8 \times 2 \times 6^8 \times (2 \times 6^8 - 1)/10 \cong 4.1 \times 10^8 \text{ sec} = \text{about } 4800 \text{ days.}$$

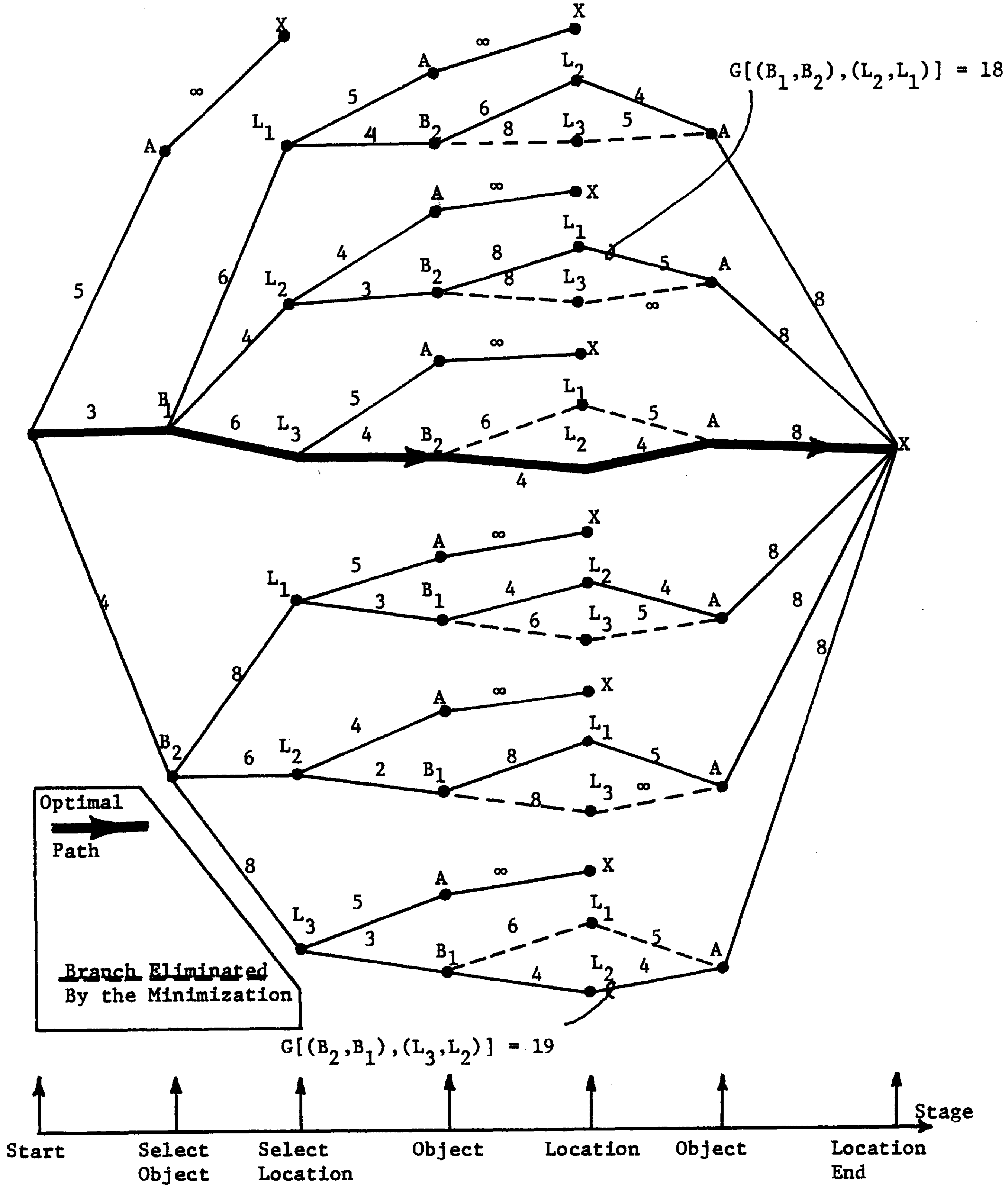

G[(B$_1$,B$_2$),(L$_2$,L$_1$)] = 18
G[(B$_2$,B$_1$),(L$_3$,L$_2$)] = 19
Optimal Path
Branch Eliminated By the Minimization
Start
Select Object
Select Location
Object
Location
Object
Location End
Stage

Figure 14

TREE CORRESPONDING TO SOLUTION OF THE BLOCKED DOORWAY PROBLEM IN FIGURE 13, USING EQUATION (9)

Now that is a whale of a difference, even when the physical space is quantized to only 36 points. Both figures would be somewhat lower if we used the adaptive processing methods discussed in the previous chapter. As yet there are no speed estimates for algorithm A*, but we may expect it to be faster than Ford's algorithm even with adaptive processing.

Some Remarks on Language

As yet we have said nothing of the computing language required to implement such interactions. Obviously it must be able to receive function definitions, preferably recursive functions. This should be possible on line, in real time. LISP 1.5[19] and TRAC[25] are two interpretive computer languages whose applicability should be investigated.

The reader has probably noticed the similarity between the first definition of Switch (equation (5)) and the way one might explain switching to a young child, not by describing the desired result but rather by showing him how, in terms of "Pick-Put," which he already knows. This seems a most promising way to develop a machine capable of manipulating: by building up its competence layer by layer, appealing to extant layers when defining a new one. First, basic motions plus grasping and carrying named objects to named locations. Then "Pick-Put." Then direct functions such as Take, defined on "Pick-Put." Then more complex functions defined on Take. Then definition of repetitive tasks by recursive functions, together with some optimization. (The latter, curiously, build graphs on which shortest paths are sought by algorithmic search. The arc lengths are found from shortest path lengths on still other graphs.) In this way, a machine may be made to "understand" rather complex task statements.

We have found it useful to think of "understanding" somewhat mechanistically. The machine can "understand" the commands when it can <u>translate</u> them from the operator's input language to a more precise mathematical language. In our case, this mathematical language is that of linear vector spaces. The state space (with no lines deleted) is the space spanned by the command set, when the latter is thought of as composed of the basis vectors* for the space. (We complicate matters beyond simple linear algebra by making the space finite, deleting lines internally, making some lines one-way, and allowing, at some places, elementary commands like carry and push which cannot be thought of as additional basis vectors.) A task is then a vector difference between the current state and the desired state. When the machine can translate an input command into a desired state, then it has "understood" the command. From there, mechanistic procedures find a good, correctly ordered, linear combination of the vectors in the command set which add up to the desired vector difference. Altogether, the machine has translated the command from the user's language into a series of commands in its own language. Extensions of the machine's manipulatory sophistication can be achieved

---

* A set of basic vectors is a linearly independent set of vectors from which any vector in the space may be composed by an appropriate linear combination.

linguistically from this point, since we have equipped it to receive descriptions of "how," at a non-trivial level.*

It is important to realize that these methods cannot teach the machine to do any task which requires motions not contained in the elementary set. In such a case, we must construct the lacking elementary command from scratch. This could be necessary especially following some manipulation mishap, when the operator may not be able to say what he wants done in <u>any</u> symbolic language, much less the restricted manipulation language at his disposal. At such times, analogic input (joy-stick or oscilloscope-light pen, for example) would be very useful.[41] Such inputs could be used to enter a new elementary command definition (although MANTRAN (Barber, op. cit.) might be better for this) or to undo a messy situation once only, under direct manual control. The point is that higher level languages are only as good as the lowest level language into which commands are ultimately translated. When these elementary commands will not suffice, no amount of linguistic window dressing can make them suffice.

General Remarks

The effort of the above linguistic discussion is directed toward giving the operator more powerful commands. Command power is

* The operational definition of "understand" as "ability to translate" is not restricted to manipulation. Polya[33] says that a student understands a problem when he can state it in his own words, i.e., translate from the teacher's English to his own. Computer programs which answer questions often do so by translating the question into mathematical set theory.[43]

a relative quantity: a command at level X is more powerful than commands at level Y if one of the former translates into many of the latter. This kind of power gives obvious advantages to the operator, who says less and enjoys it more. But it is not free. There must exist a translator capable of deducing the correct commands at level Y which correspond to some command at level X. The latter may be the result of some higher translation, while the former may require still further translation. This is pure subgoal seeking and has its counterparts in artificial intelligence problems. At some X levels, however, either or both of the following difficulties may arise:

1) The translational effort may swamp the computer. This happened in the blocked doorway problem of Figure 13 when formal solution methods were tried. In such cases, the operator must intervene and supply at least sets from which subgoals are to be sought, plus a procedure for seeking them. Hence the function definition techniques. Recursive functions can afford very terse communication. Figure 15 represents the qualitative relationship between command power level and unaided computational load.

2) The commands may become so rich that they are difficult for the operator to use. They may involve many objects and conditions, perhaps long sequences of moves. The operator may have to supply many parameters or subgoal sets. This may force him to plan motions farther into the future than he feels the manipulator can work without mishap. Or he may not be easily able to forsee the many consequences implicit in such a long motion sequence, and might

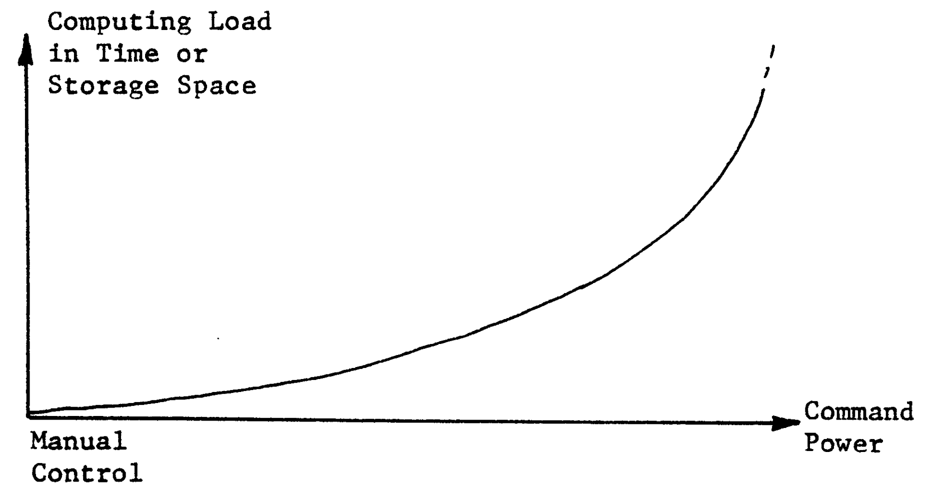


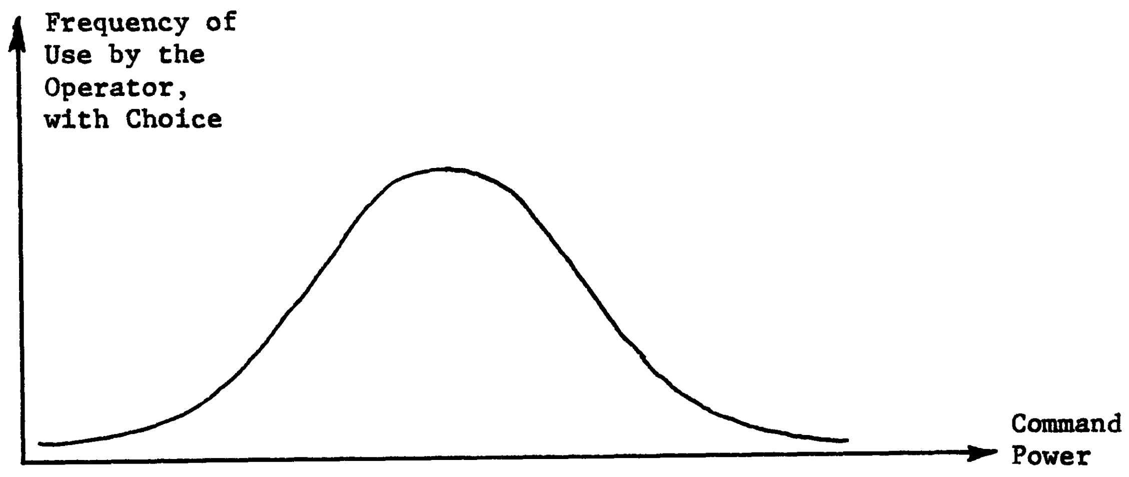
Figure 15

QUALITATIVE RELATION BETWEEN COMMAND POWER LEVEL AND UNAIDED COMPUTATIONAL LOAD

decline to commit himself so far ahead. We may speculate that the maximum the operator would tolerate will be of the order of ten or less "Pick-Put" operations, more or less depending on the dexterity of the manipulator.

Rich commands have the advantage/disadvantage of being very good for exactly one task, and no good for anything else. Since each may require the operator to remember, forsee, and communicate a lot, it is likely that he will use fewer commands at higher levels, more at median levels, and fewer again at levels approaching manual control. This is depicted qualitatively in Figure 16.

Figures 15 and 16 tend to concentrate man-machine communication at a level between manual control and full computer execution of lengthy and complex tasks. More work should be done to



manual control--
frustrating and
demanding of
attention during
execution, but
very general.

rich symbolic--
demanding during
framing of the
command. Long
computing time.
Long committment
by operator, with
uncertain outcome.
Highly specific.

Figure 16

QUALITATIVE RELATION BETWEEN COMMAND LEVEL AND FREQUENCY OF USE BY OPERATOR

see where this region lies. Its location will be influenced by the variety of input modes the operator can use, the power of the computer, the amount and type of specialized tools available to the manipulator, the hostility of tasks and environment, and the ability of the operator to keep his head. He will almost certainly appreciate a command if its computation takes even as long as he would have to work manually to accomplish the same manipulation. Longer time delays will probably push the hump in Figure 16 further to the right.

Back of all this, we have some notion of the heirarchies of decision and capability needed to manipulate. They are listed in Table 1, in descending order of anticipated difficulty for the computer-manipulator. (It is assumed that if the machine can operate at level N, it can operate at all levels below N in difficulty.) We contend that the operator must be able to instruct or aid the manipulator at all these levels. We have mentioned above that it may not be desirable for the computer to essay above certain levels, corresponding roughly to level 3 of the Table, on which great effort is currently being expended by Minsky and his group. Ways of accomplishing portions of level 4 have been outlined in this chapter. The lower levels have for the most part been demonstrated by this author on a plotting table manipulator, about which more is said in the next chapter.

Summarizing this chapter, we have shown that complex tasks can be treated by graph methods if we can find ways to represent these problems using strings of low-dimension graphs instead of one large graph. The analogy to stringing the elementary commands together should be obvious. A direct way of making graphs composed of other graphs was derived, and its relation to the operator's input language discussed. We speculated that commands could be too powerful to be convenient, and that the operator might prefer to work at a less sophisticated level. He will certainly desire access to manual control to handle situations not covered by his input language at higher levels.

Table 1

DECISION AND CAPABILITY LEVELS

1. Generate main goals of the entire task (such as "Take apart the typewriter").

2. Generate intermediate general configurations of objects, tools and obstacles (this would correspond to the repair manual for the typewriter).

3. Recognize an object, tool, or obstacle on sight or contact.

4. Designate an object and generate a new location or orientation to a depth of K recursions (e.g., K objects must be moved before a condition is satisfied--find the order in which to move them, etc.)

5. Find a way to move a designated object to a designated location or orientation, if a way exists which does not involve moving or touching other objects.

6. Recognize that 5) is impossible in some case.

7. Distinguish (for planning purposes only--no recognition required) the designated object from others.

8. Maneuver the jaws through an unknown environment and acquire information about the locations and sizes of the objects lying therein.

9. Maneuver the jaws through a known field of objects, including generating the motion plan in detail.

10. Analyze on-off and graduated touch sensor information, such as contact, tight grasp, etc.

11. Detect that the jaws have reached a designated location.

12. Energize a designated manipulator prime mover and measure how the jaws' location changes.

CHAPTER VI

A PHYSICAL DEMONSTRATION OF STATE SPACE MANIPULATION CONTROL

The ideas described in Chapters II through IV have been demonstrated on a three-degree-of-freedom manipulator converted from a plotting table. Square objects may be grasped and moved about in a region 15 inches on a side. No jaw rotations are possible. The two computers shown in Figure 3 of Chapter I are condensed into one PDP-8. No transmission delay is introduced. See photos. Using this apparatus, one can arrange the objects in patterns and conceivably make the moves required in board games like chess or checkers, although the operator would originate the moves! The introduction of transmission delay would add nothing essential, since the operator is not involved in control of the jaws during task execution. The delay could as well be an hour as a second (the time presently taken to apply the algorithm and extract the path), with no difference in performance.

The apparatus was intended to show what could be done with a small computer and a modest state space model. As has been shown, when many objects are involved, large state spaces can result. We limited ourselves to the state vector $(x, y)_{\text{Jaws}}$. A jaw status

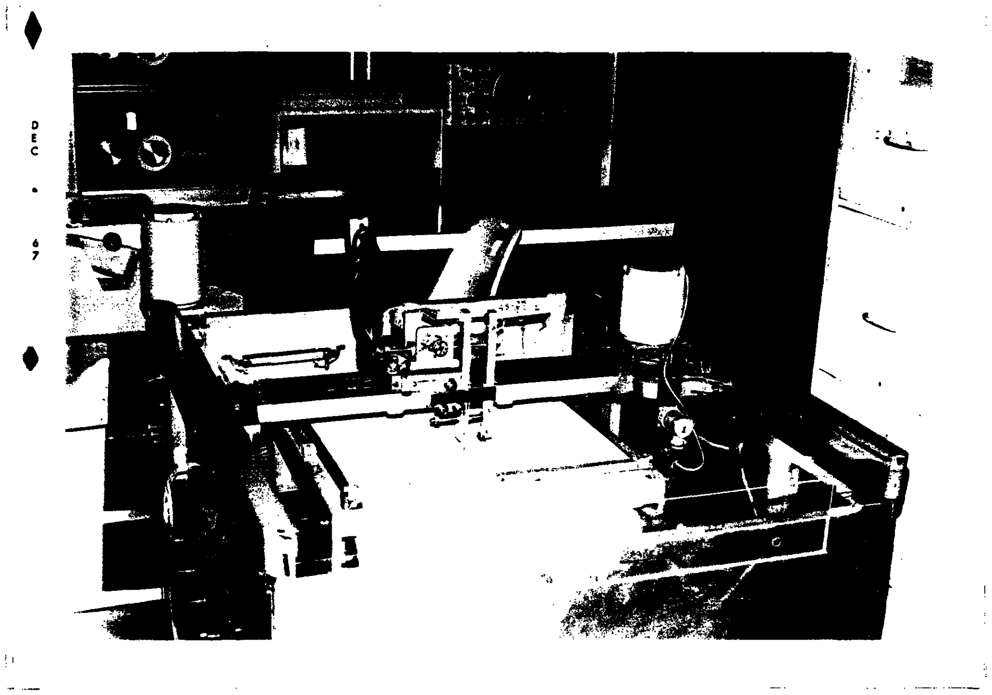


Figure 1

PLOTTING TABLE MANIPULATOR WITH PDP-8 COMPUTER IN THE BACKGROUND

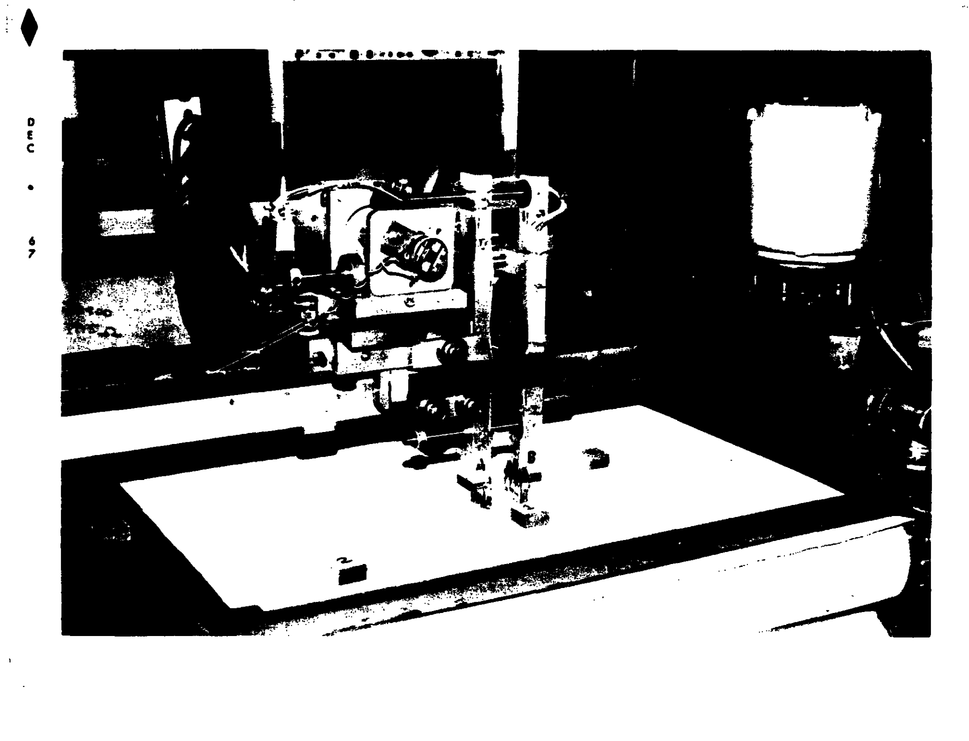


Figure 2

CLOSER VIEW OF MANIPULATOR, SHOWING JAW MECHANISM

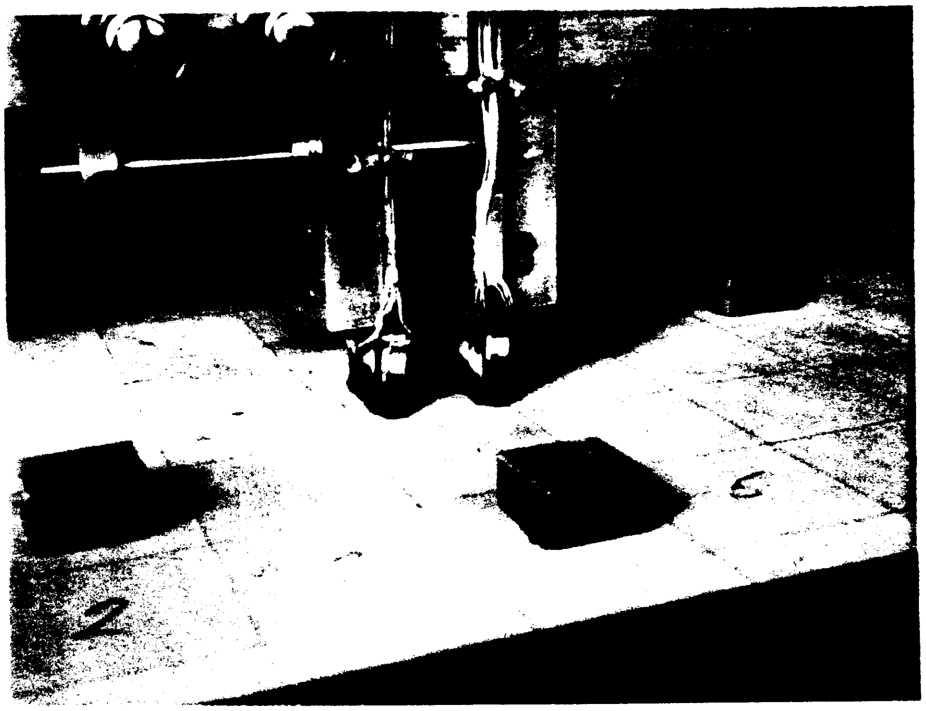


Figure 3

CLOSE-UP VIEW OF JAWS SHOWING TOUCH SENSORS.
THREE OBJECTS ARE NEARBY.

variable similar to H (see Chapter II, page 49) was carried separately as a single parameter to indicate whether the jaws were empty or full. This cut memory requirements by 50% but put considerable constraints on command power. The resulting state spaces looked like Figure 7a or 7b of Chapter V, depending on the value of the jaw status parameter. Diagonal moves were allowed.

The PDP-8 computer is quite fast, having a 1.5 μs cycle time. Ours has 4096_{10} words of 12 bit core memory. The programs to implement the demonstration take up about 2000_{10} words, the state space another 450_{10}, plus 50_{10} reserved for writing the path when it is extracted.

Three Slo-Syn stepping motors (Superior Electric Co., Bristol. Conn.) drive the three degrees of freedom. The motors index 1/200 of a revolution each time a computer clock pulse is gated to a pulse amplifier. Pulleys transform each such pulse into 1/100 of an inch motion of the jaws. After forward or reverse is selected, a program to move the jaws a known distance need merely count out the correct number of pulses at any desired rate up to about 200 pulses per second. A third motor opens and closes the jaws in the same way. The result is repeatable discrete displacement. For this reason, no feedback is sent from the motors to the computer.

All communication between operator and computer is via teletype. The operator can give objects symbolic names, or designate locations by their coordinates. (Allowing the operator to give locations symbolic names is a trivial extension.) He can also designate certain objects as walls, after which the computer will consider them immoveable and reject commands to move them.

The computer communicates with the manipulator's environment by means of touch sensors on the jaws and limit switches on each degree of freedom. There are three continuous pressure sensors on the gripping face of each jaw. In addition, ten on-off contact sensors are arrayed on the outsides of the jaws, as shown in Figure 4. Each on-off sensor is about 1/8" wide and each graduated sensor, made with pressure sensitive resistance material, is about 3/16" wide.

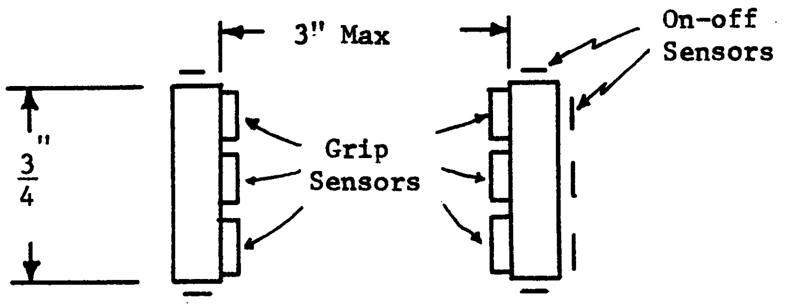


Figure 4

ARRANGEMENT OF SENSORS ON JAWS, TOP VIEW. NO SCALE

Details of these sensors' construction and performance appear in Appendix II.

The on-off sensors allow the computer to detect collisions with unknown objects. Following collision, the object's coordinates are estimated by the computer in two steps. 1) The jaws' location is found from the value of the last successfully occupied state plus how many pulses were issued in which directions beyond that state prior to the collision. 2) The sensor pattern obtained during the collision is compared to several standard patterns to help the computer decide whether the object is directly ahead, off to the left, or to the right. This works best when the jaws are moving parallel to a row of on-off sensors (Figure 5a), and less well when they are moving diagonally (Figure 5b). In particular, the situation in Figure 5b often results in no sensor being touched until the object has been pushed aside. Corner sensors facing diagonally would help. The computer then types out the expected coordinates of the object and asks the operator to give it a name. The computer

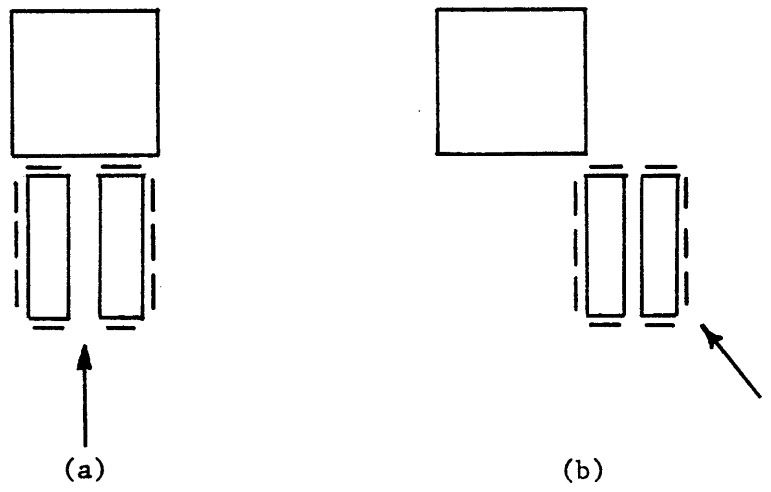


Figure 5

DETECTION OF AN OBJECT DURING COLLISION

thereafter keeps track of the object in a table, and the operator may refer to it by name in subsequent commands.

The graduated sensors are used by the computer to detect when grip is tight, so that closing of the jaws can safely stop, and to detect when grip is loose so that it can be retightened. A potentiometer on the jaws shows the computer whether the jaws are wide open, grasping, or shut tight. Then the computer can tell if it really grasped anything or not, and if so, how big it is. These remote feedbacks give the apparatus some independence from the operator so that it can check performance against the plan or react quickly to a collision in spite of delay between computer and operator.

When the program is first loaded into the computer, it has no knowledge of objects or walls. When the operator names an object and gives its coordinates, these are entered in the Object List. (After a collision, the computer gives the coordinates and the operator gives only the name.) Objects given the name W (wall) are entered in the Wall List. In these ways, the computer accumulates and classifies knowledge about the environment. Each time the operator calls for a move, the computer, using the data in the two lists, constructs the graph from a blank grid by removing lines corresponding to objects and walls, after which the graph looks like Figure 7a or 7b of Chapter V.* Paths corresponding to the operator's commands for jaw motion are found directly via the Ford algorithm. Commands to pick up or carry an object are deduced via the command perturbation method, described in connection with Figures 1 through 5 of Chapter V. The existing programs are therefore poised for

---

* Thus the graph's information is really in the form of lists. For this reason and because of the recursive features of manipulation, we have cited list-processing languages like LISP and TRAC for future application.

implementation of simple functions like Take, although this has not been accomplished yet.

What the Demonstration Taught

The author learned many things from this demonstration, aside from the general perversity of inanimate entities:

1) A lot of basic manipulation can be accomplished using only the state vector $(x, y)_{\text{Jaws}}$ and a jaw status variable. This conclusion, plus the threat of huge state spaces, prompted the chain graph study reported in Chapter V.

2) Inanimate entities are hard to talk to. Once we limit ourselves to a small manipulation alphabet of static, finite motion commands, we are limited in what we can say to the manipulator. Other statements will not be "understood". This realization, plus the restricted state vector, led to the remarks on language in Chapter V.

3) A lot of manipulation tasks and manipulation strategies can be expressed recursively. We demonstrated this property of the strategies with the recovery procedure following collision with an unknown obstacle: Say a path $\{u_1, u_2, \ldots, u_n\}$ was interrupted by collision during execution of command u_i, $1 \leq i \leq n$. First, command u_i is inverted and undone exactly the number of pulses which had been issued during the attempt to execute it. In principle this returns the system to the last successfully occupied state. (See 4) below.) The computer then types out its estimate of the object's coordinates. The new object is named by the operator and is entered

into the appropriate list. The original goal state remains unchanged, so the computer simply treats the present state as it did the initial state and starts over. It finds a path between these states, using the new information, and motion continues without any intervention by the operator, except to name the new object.

Recursive properties of tasks appear in the blocked doorway problem or in building a tower:

A tower (N) blocks high =

A tower (N-1) blocks high + one block on top,

N = 2,3,...

A tower (1) block high = one block on the table.

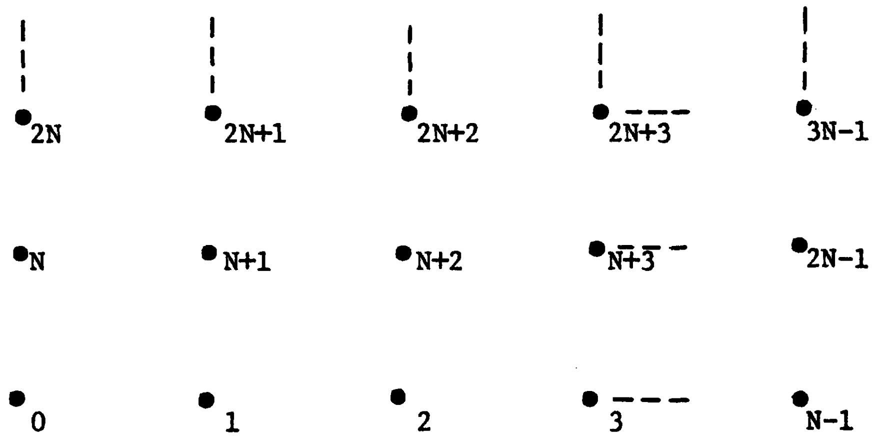
Thus "A tower(N) blocks high" is a recursive function of N. The same may be said of a row extending from the left, say, a circle going clockwise, or a pyramid, for example.

4) If an object has been pushed aside due to being missed by the touch sensors, then the object is in a configuration which corresponds to <u>no</u> jaw state. The input language is of no use at this point. This sort of mishap emphasized the need for access to more primitive commands when trouble occurs. In this program, the operator can call explicitly for elementary commands, which he can halt in progress in order to guide the jaws to the displaced object. This amounts to manual rate control. Touch sensors remain active throughout.

The author wrote another program early in his research, which allows more flexible commands at elementary levels. The operator may designate one of 8 directions and the speed of the jaws via teletype. Speed or direction may be altered in steps any time during motion, which otherwise continues until halted by collision or receipt of another command. Touch sensors may be inactivated to facilitate pushing. The author found that with some practice, he could perform the test task reported by Ferrell (reference 11, page 59.) in about two minutes, under a four second time delay. Completion time was somewhat limited by the maximum jaw speed (about two inches per second) and by the awkward command mode. It seems clear that the operator, when he needs to intervene, should not be restricted to the same set of commands that the computer uses for state-space planning, but should have access to the best combination of analogic and verbal input modes for situations where higher level commands will not suffice. Search routines such as those devised by Ernst (op. cit.) should also be available.

5) Many of the computational problems associated with implementation of algorithms were discovered while programming the demonstration. Algorithms are usually stated procedurally, saying in effect, "Just shut your eyes and do as you're told." Little attention is paid to strategies for applying the algorithms (except for A*, which contains its own application strategy), and no one mentions multipath solutions or other such troublesome items.

A most intriguing computational problem is storage of a multi-dimensional space in a one-dimensional memory (core, disc or tape). If we give the points on a grid linearly ascending subscripts, as in Figure 6, where part of a two-dimensional grid is shown,



N = number of points on a side.

Figure 6

SUBSCRIPTS ON GRID POINTS

then we may associate the points sequentially with memory registers. Clearly, the four neighbors of an interior grid point associated with register number M are in turn associated with registers M+1, M-1, M+N, and M-N. If the grid is three-dimensional, two more neighbors are at $M+N^2$ and $M-N^2$. Corresponding relations may be derived for diagonal neighbors and for neighbors in a higher dimension space.

To find the (x, y, z,...u) coordinates of a point in a space with N points per side, given the value of M, we need only recognize that

$$M = x + Ny + N^2z + \ldots + N^Ru, \quad R = \text{dimension of the space}$$

That is, M is a radix N number or equivalently a polynomial in N. (Compare this for N=10, for which M is a decimal number, or N=8, for which M is octal, etc.) Then conventional number-base conversion methods will extract x, y, z, ... from M, given N. A similar formula may be derived when there are N_x points on the x axis, N_y points on the y axis, etc.

The significance of the relations between M and the locations of the neighbors is that points adjacent on the grid may be far apart in linear memory. Yet, to apply the algorithms, we need rapid access to all of a point's neighbors. If we do not have rapid access to all these points (the case if they are stored on magnetic tape) then we may have to wait intolerably long while data at "neighboring" points are retrieved. Care must be taken to arrange data for rapid access in such structures. One might use more space and store at a point all the data associated with its neighbors. One might store the points at random in linear memory, instead of according to radix N numbers. Some day we may have large associative memories, allowing us to call into an arithmetic unit all points tagged as "adjacent to (x, y, z, ...)," with no reference to M+1, M+N, and no need to calculate them.

Film Record

We have made a short film of the apparatus in action. It demonstrates naming of objects, moving the jaws to a location, grasping a named object, recovery from collision with an unknown object and remembering it, and recovery from collision with an extended wall, whose location is also remembered. The object discovered by collision is later grasped and carried on command, and the wall similarly discovered is avoided in all subsequent motion. Not shown in the film is the apparatus' negotiation of the maze discussed in Chapter IV.

CHAPTER VII

A LOOK INTO THE FUTURE

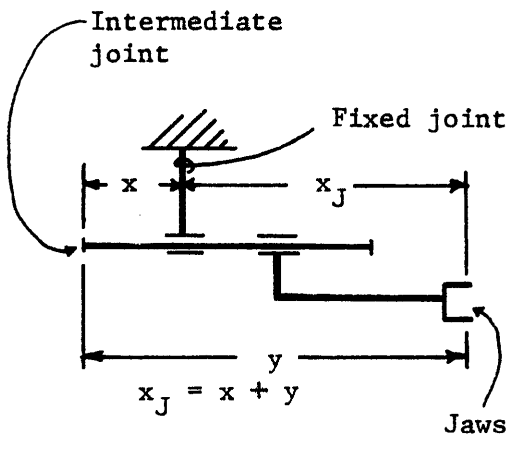
What will supervisory controlled manipulation be like five years from now? We can make some fairly reliable predictions, based on our work and that of others. These predictions may be thought of as suggestions for further work.

The operator will work from a console which has a television screen, a joystick model of the manipulator, a light pen, a box of switches, and a teletype or microphone. The TV and light pen go together, the former simultaneously showing real scenes from the task site and a line drawing generated by the computer to simulate such scenes and illustrate its model of them. The operator may give commands with his voice or by teletype, or by using the light pen to sketch paths on the TV screen, indicate objects and their new configurations and designate regions where certain commands are likely to be applicable, others inapplicable. Some regions may be given fine quantization, others ignored, hence forbidden to the jaws because they are dangerous, delicate or as yet unexplored. With the buttons, the operator may input symbolic statements like Push, Screwdriver, or Stop. With the joystick he can assume manual control or indicate to the computer an orientation he wants the manipulator to assume.

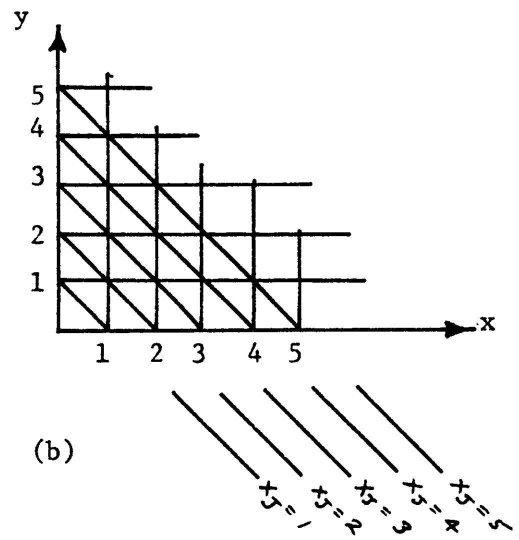
From command and sensor data, the computer will build state spaces which differ from those we have discussed in three ways. First, quantization will be non-uniform. Second, some commands with special utility, like Push, will be allowed only where (in time and space) the operator has hinted at their usefulness. This will save processing effort. The biggest difference will be that these models will not correspond so closely with the actual task site as those we have discussed. Objects will not have to be exactly the anticipated size, or be located only at certain points. The approximate nature of the resulting paths will be counterbalanced by more sophisticated execution routines with built-in search features, aided by high quality touch and vision sensors. A sensor which can detect shear forces will be especially useful for finding edges and detecting a slipping grasp.

The manipulator will have a redundant structure. This will require more sophisticated models. To give an idea of what these models will be like, we show in Figure 1b the state space for the two degree-of-freedom manipulator in Figure 1a, which works in a one-dimensional task space. The jaws are supported from the intermediate joint by a slider bearing, while the intermediate joint is in turn supported from a fixed joint via another slider bearing. Coordinate x locates the intermediate joint with respect to the fixed joint, coordinate y locates the jaws relative to the intermediate joint, and x_J locates the jaws with respect to the fixed joint. The main thing to notice is that many points in the state space correspond to the same jaw location. Algorithm A^* (see Chapter IV) is ideally suited to such spaces, since processing will stop as soon as any point in the terminal manifold corresponding to desired jaw location is reached.

Redundancy has one main purpose: to allow the manipulator to reach around things. For this reason, the locations of the redundant elbows are important throughout jaw motion. Consistent



(a)



(b)

Figure 1

SIMPLE REDUNDANT MANIPULATOR AND ITS STATE SPACE

with our previous ideas, this means that redundant elbow locations must be made state variables, so that forbidden elbow locations can be expressed. Interpolation will have to be used to express forbidden locations of points between elbows in terms of elbow locations. If reaching around is not necessary at all in some environment, one saves vast amounts of computation by freezing the redundant joints. It does not make sense to have a redundant manipulator in an unconstrained space, because one usually ends up employing some elegant method to throw the redundancy away.

Adding a lot of elbow state variables will obviously do the same harm as adding object state variables. A heuristic solution is to plan jaw motions first, ignoring the elbows, concentrating on task constraints. This will involve state spaces like those we have considered in Chapters II through V. Then we switch to a state space like Figure 1b, which shows no task constraints but rather might have areas blanked out to show regions forbidden to the elbows, as in Figures 2a and 2b:

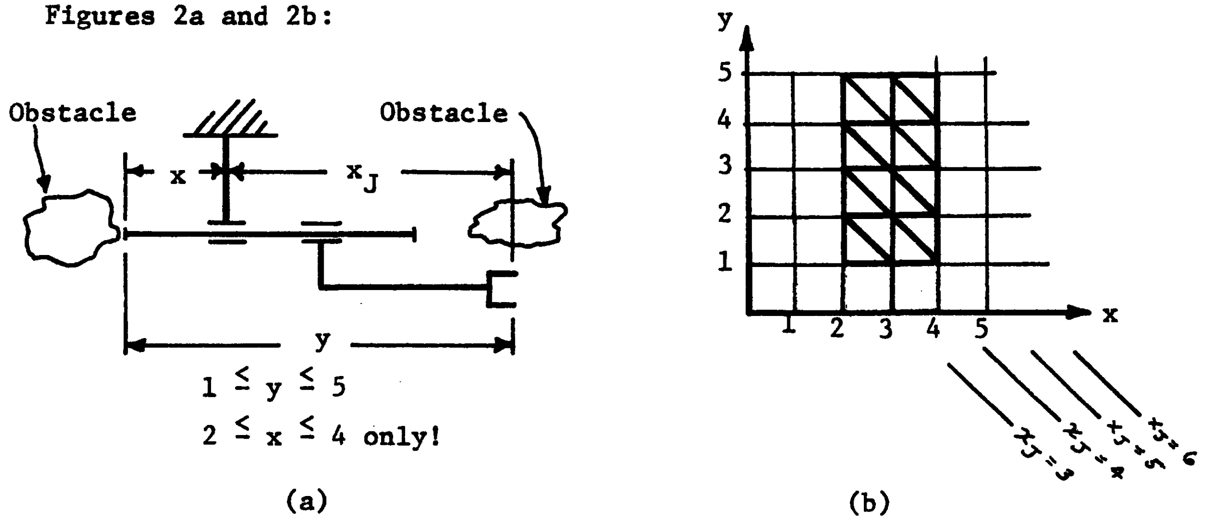


Figure 2

STATE SPACE FOR ELBOWS WITH ONE ELBOW VARIABLE RESTRICTED BY OBSTACLES

In this space we plot a path for the elbows, given that the jaws must follow a specified path in physical space. (Say, move x_J from 5 to 2 in Figure 2b.) Of course, portions of the jaw path may not be consistent with elbow constraints, necessitating replanning of those portions of the jaw path. The operator may be able to aid planning of the elbow path, using his model of the manipulator to input intermediate configurations which the remote manipulator should occupy or approximate during its motion. While this will help, the planning of constrained elbow motions remains a formidable problem in conventional control.

The operator of a future Supervisory Controlled Manipulator will have a versatile input language available, which he may use via teletype or voice. This will include functions like those described in Chapter V, routines like those possible in MANTRAN[2], and statements more like English. Designers of such languages must take care that the allowed commands are not too powerful, lest they become thereby so specific that the operator must keep a large assortment of them in mind in order to do general tasks. There must also be available some less powerful but more general commands as well.

The computer must also have some facility for learning how to manipulate more skillfully. Using defined functions, the operator can "teach" the computer various routines. But more autonomous learning behavior might be very desirable, allowing the computer to take its cues directly from the manipulator and the task site. Appropriate search strategies or useful sensor patterns are among the things which might be learned in this way.

CONCLUSIONS

1. A formal structure has been developed in which manipulation tasks may be modelled. This structure consists of a discrete state space for the task site, expressed as a finite graph. Such models have been shown capable of expressing the logical and physical constraints necessary to describe such tasks as dodging obstacles, grasping and carrying objects, pushing objects, and some complex ordering and decision problems.

2. The use of such models enables the human operator of a manipulator, however remote it may be, to issue commands at a goal level, leaving methods and execution to a computer, which maintains the state space, receives commands, and operates the manipulator with the aid of a smaller computer at the task site.

3. The resulting system is called a Supervisory Controlled Manipulator, in which the operator performs those parts of manipulation, the decision-making and pattern-recognizing tasks, for which he is best suited; the computers carry out the routine work, planning the details of task execution, monitoring sensors, and reporting progress or difficulty to the operator. A simple demonstration of these ideas was built and operated.

4. The more carefully a task is planned, the less necessary is feedback during execution. The use of small amounts of feedback thus requires great detail in the task model, which takes its toll

in computer time and space during task planning. On the other hand, computational load during planning can be relieved by omitting details from the model and executing the cruder plan more carefully with more attention to feedback. This spreads some of the computational load onto the execution phase without degrading the system's performance.

5. The state space approach has the advantages of being simple, direct, and capable of generalization to a wide variety of tasks or to manipulators with redundant structure. The operator can impose most any criterion of optimality he desires onto the computation, greatly influencing the nature of the solution paths, and allowing the operator to adapt the method to varying degrees or areas of risk, knowledge, confidence, and so on.

6. The simplicity of the method leaves it prey to certain inefficiencies: the computer takes time to investigate task possibilities which have no chance of being solutions, and uses space to store these possibilities on the chance that they might be applicable in some other situation. Yet it appears that heuristic procedures, which might be less troubled by such inefficiencies, are not well suited to problems at the simplest level in manipulation. Rather, systematic methods, such as the state space model, may be better matched to simple problems, with heuristic methods reserved for higher planning levels.

7. Much of the time and space inefficiency of state space models can be eliminated by providing the operator with an input command language which includes recursive functions, with which he can build spaces to order which are _ad hoc_ to a given task. Heuristic methods may also be able to build simple state spaces in the same way.

Thus the state space method can be part of a man-machine system or an autonomous system for manipulation.

8. A number of search algorithms exist for finding the optimal solutions, among them conventional Dynamic Programming, the Ford algorithm (which we showed to be a form of Dynamic Programming), and the recent Hart-Nilsson-Raphael algorithm, (algorithm A*) which is more efficient than the others.

APPENDIX I

MATHEMATICAL STATEMENT OF GRAPH THEORY AND REMOTE MANIPULATION

Let X be a set of points called states. Let $C = \{c_1, c_2, \ldots c_k\}$ be an ordered set of one-one functions called commands which map X into itself as follows: For each $x \varepsilon X$ let A(x) be the set of states adjacent to x:

$$A(x) \subseteq \bigcup_{i=1}^{k} c_i(x) \qquad \text{with } x \notin A(x) \text{ and } c_i(x) \neq c_j(x) \text{ for } i \neq j \qquad (1)$$

We assume that for most x and $y \varepsilon X$

$$y \varepsilon A(x) \Leftrightarrow x \varepsilon A(y)$$

that is, for each i, $1 \leq i \leq k$, there is a $j \neq i$, $1 \leq j \leq k$, such that $c_j = c_i^{-1}$. We define the composition of commands $c_j c_\ell(x) = z$ by

$$c_\ell(x) = y \text{ and } c_j(y) = z \qquad (2)$$

Then

$$c_j c_j^{-1}(x) = x = c_j^{-1} c_j(x)$$

although

$$c_i c_j \neq c_j c_i$$

in general. Then we have implicit in C an identity command I:

$$I(x) = x \qquad \text{for most } x \varepsilon X$$

If $I(x) = x$ for all $x \epsilon X$, and all of the above held, we could say that the set C of commands and the operation of composition formed a non-commutative group. However, there are interesting situations in which a command could take state x to state y and either no single c_i could take y back to x or no finite string of c_i's could restore x at all. The former could occur if an object were dropped some finite vertical distance. The latter could include driving home a nail or any thermodynamically irreversible process. Thus it is possible for some x and y to exist such that $c_i(x) = y$ exists but $c_i^{-1}(y) = x$ does not.

A task specification consists of a pair of states (x,y), x being the current state and y a desired future state. A human operator presumably picks y for reasons of his own. A procedure for accomplishing this task consists of a string

$$c_a, c_b, \dots c_\ell, c_m \tag{3a}$$

such that

$$y = c_m c_\ell \dots c_b c_a(x) \tag{3b}$$

in which the c_i are to be selected from C with replacement.

Our problem is to design a procedure by which a computer can deduce the sequence (3a) from (x,y), X, and C. A way to do this is to let (C,X) define a graph G such that a directed branch exists from x_1 to $x_2 \epsilon X$ if $x_2 \epsilon A(x_1)$, and an undirected edge exists between x_1 and x_2 if $x_2 \epsilon A(x_1)$ and $x_1 \epsilon A(x_2)$. Next, for each x, we associate with x and $c_i(x)$ a positive non zero metric (cost, distance, risk, etc.) called $u_i(x)$ defined on each transition from x to $c_i(x)$ for each i.

We presume without loss of generality that the operator wants procedures which minimize this metric while carrying out the tasks he specifies. We must then find minimum-metric paths (sequences of adjacent states) in G from x to y in order to generate sequences like (3a). This is true because associated with a given path there is exactly one sequence of commands from C which, applied in order, will result in exactly that sequence of states being occupied in exactly that order. (Proof is trivial)

To put this plan into operation, we need first a shortest path algorithm, such as that of Ford. Second, we need a function defined on pairs of states (x,y) such that

$$f(x,y) = \begin{cases} c_i & \text{if } y = c_i(x) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

(f(x,y) = 0 if x and y are not adjacent.) Applying f(x,y) in order to the adjacent states in a path, starting at the beginning of the path, we obtain a sequence like (3a).

One way to set up our notation for X which gives a simple f(x,y) is to consider X to be a k/2 - dimensional (discrete) vector space, so that x is a k/2 - vector. (k is the number of commands in C and will be even if commands usually have inverses.) Then the state $c_i(x)$ is represented by a k/2 - vector whose i^{th} element differs from the i^{th} element of x by unity, say. Then the functions $c_i(x)$ are simply

$$\begin{aligned} c_i(x) &= x + \varepsilon_i && \text{for } 1 \le i \le k/2 \\ c_i(x) &= x - \varepsilon_{i-k/2} && \text{for } (k/2+1) \le i \le k \end{aligned} \tag{5}$$

which implies

$$c_i = c_{i+k/2}^{-1} \quad \text{for } 1 \leq i \leq k/2$$

whenever the inverse is defined. In (5), ε_i is the i^{th} fundamental basis vector of a k/2-dimensional vector space. This allows us to define f(x,y) as

$$f(x,y) = \begin{cases} y - x & \text{if } y - x = \pm\varepsilon_i \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

This in turn allows us to code the c_i as

$$\begin{aligned} c_i &\to \varepsilon_i & 1 \leq i \leq k/2 \\ c_i &\to -\varepsilon_{i-k/2} & (k/2+1) \leq i \leq k \end{aligned} \tag{7}$$

and to compute quite easily the state $c_i(x)$ for any i and x, given only i and x.

<u>Remarks</u>

Actually, one may append to C many commands without increasing the dimensionality of X if such commands do not increase the number of points in X. Such commands must rather provide alternate means of making transitions between existing states. See Figure 1.

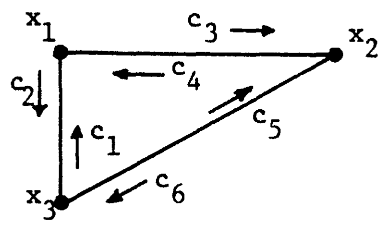


Figure 1

ADDING ALTERNATE COMMANDS

Here, C originally consists of $\{c_1, c_2, c_3, c_4\}$ and the corresponding X has dimension 2. c_5 and c_6 provide new edges but no new states. Further, c_5 and c_6 do not connect any states which were originally adjacent. Thus we preserve the uniqueness property of C:

$$c_i(x) \neq c_j(x) \qquad \text{for any } i \neq j$$

APPENDIX II

TOUCH SENSOR DESIGN AND PERFORMANCE

This appendix describes the pressure and contact sensors used on the demonstration apparatus discussed in Chapter VI. The pressure sensor is a small, continuous output device suitable for detecting low level normal forces on the fingers of remote manipulators and prostheses. The sensitive element is carbon-impregnated rubber whose resistivity changes by more than a factor of one hundred when pressed moderately hard with the tip of one's finger.

Some advantages of the design are:

1) Low noise -- less than 5% peak to peak.
2) Approximately constant <u>percentage</u> sensitivity of about 10% when employed as a force sensor (similar to Weber's Law).
3) Easy detection of displacement changes smaller than .001 inch, with a working range of about .010 inch.
4) Detection of load changes as small as seven grams with a bias load of fifty grams, or detection of fifty grams with a bias load of 450 grams. Maximum load over 500 grams.
5) Rapid initial response.
6) Good repeatability with increasing loads.

Some disadvantages are:

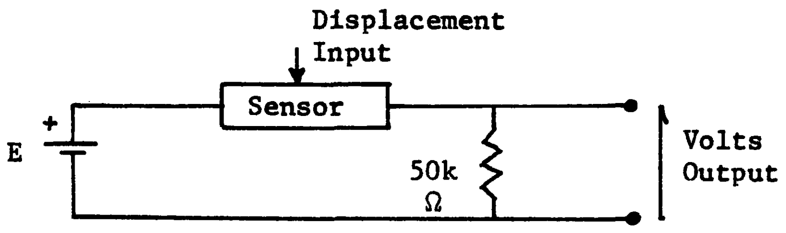
1) As much as 40% hysteresis.

2) Long time to come to equilibrium after initial response.

An approximate transfer function model, in Laplace Transform notation, is

$$\left.\begin{aligned} \frac{\text{Volts out}}{\text{Displacement in}} &= K_1 \frac{1.1\tau s + 1}{\tau s + 1} \quad \text{increasing load} \\ \text{———— '' ————} &= K_1 \frac{2\tau s + 1}{\tau s + 1} \quad \text{decreasing load} \end{aligned}\right\} \begin{aligned} &\tau \cong 5 \text{ sec.} \\ &K_1 = \text{scale factor} \end{aligned}$$

when the sensor is used in a voltage divider circuit such as



Many trial designs were rejected before a good balance of low noise and high sensitivity was reached. See Figures 1 and 2. Hysteresis seems to be a property of the rubber and could not be eliminated. Low noise was achieved by using a hard epoxy cement rather than a rubber cement. High sensitivity was obtained by keeping the electrodes very thin and by keeping all cement away from the rubber. All response tests reported here employed the voltage divider circuit above, the output being sensed by a high impedance volt meter-chart recorder.

In the design shown in Figures 1 and 2, the rubber is 1/8 x 1/8 x .050 inches, cut from material whose no-load resistivity is over one megohm-inch.* When the epoxy has dried, the excess plastic base on each side of the electrodes is trimmed off with a belt sander, and the sensitive end is wrapped with three or four turns of Saran Wrap to protect the electrodes and to keep the rubber from falling out. Pigtails are soldered to the ends of the electrodes, completing construction. The resistance of an assembled unit varies from over ten megohms no load to about 2000 ohms under firm fingertip pressure.

The following figures describe the behavior of a typical unit. Figure 3 shows the response when the sensor is held in a micrometer. Each vertical jump in output is the response to .001 inches change in micrometer setting. This Figure was used to construct the lead-lag transfer function referred to above, and to make a hysteresis plot, shown in Figure 5. Figure 4 is a sensitivity test. The two 25 gram weights are applied in sequence, and then a 6.5 gram weight is repeatedly applied and removed. This percentage load change is just about at the limit of sensitivity, but the noise is considerably smaller than the response to the load changes.

The on-off contact sensor consists of two pieces of .001 inch brass shim stock separated by a piece of paper, the whole glued together with epoxy cement. See Figure 6.

---

* Similar material is available from Coe-Myer Corp., 315 N. May St., Chicago, Illinois.

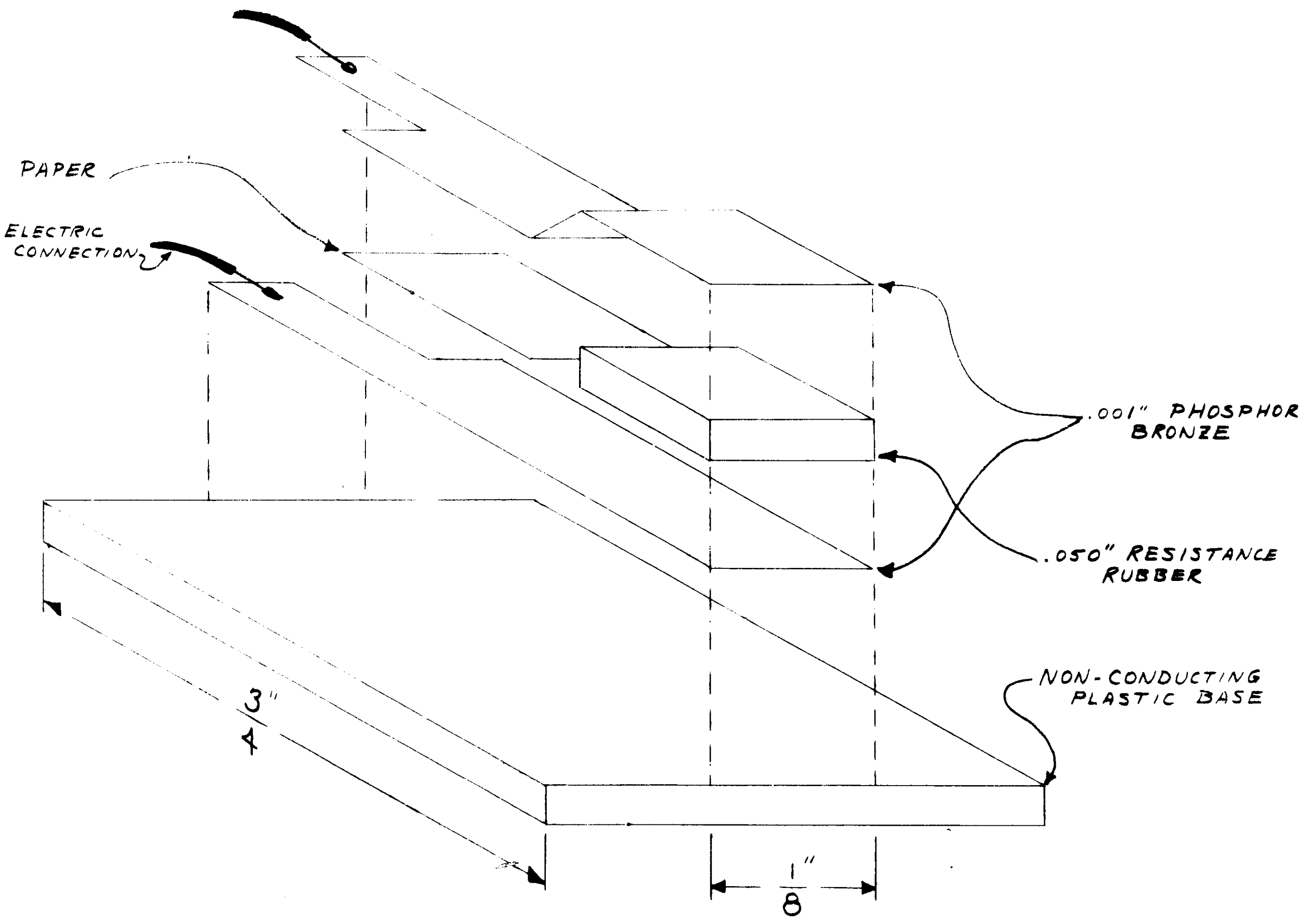


FIGURE 1 - EXPLODED VIEW

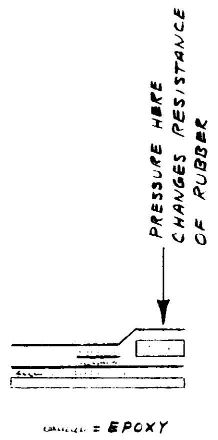


FIGURE 2

SIDE VIEW

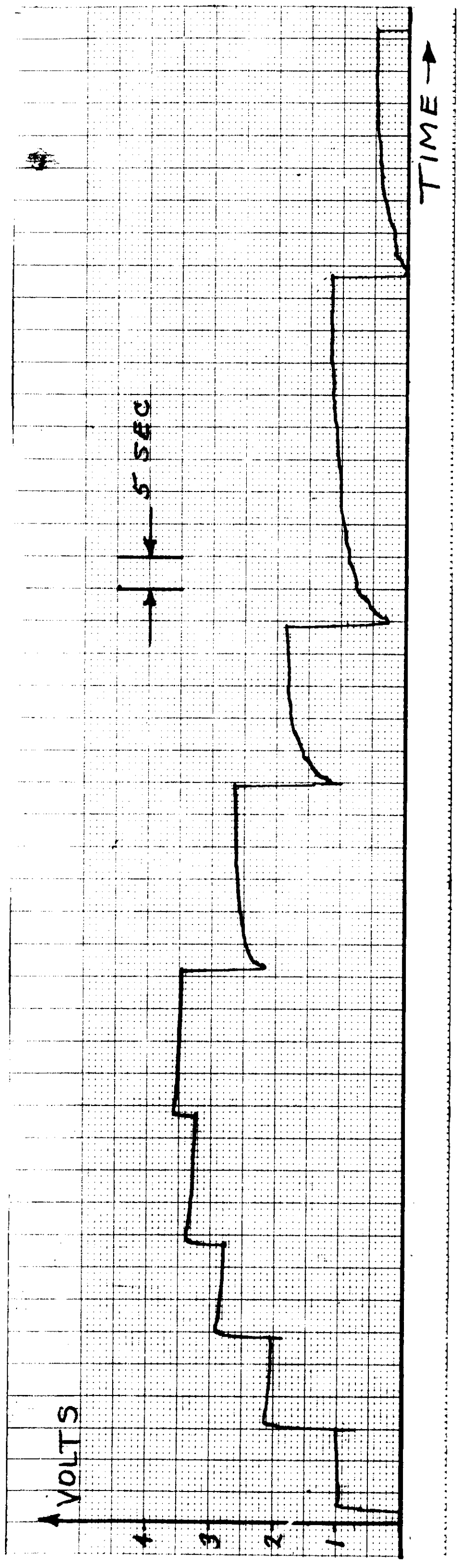


Figure 3 . Displacement Calibration

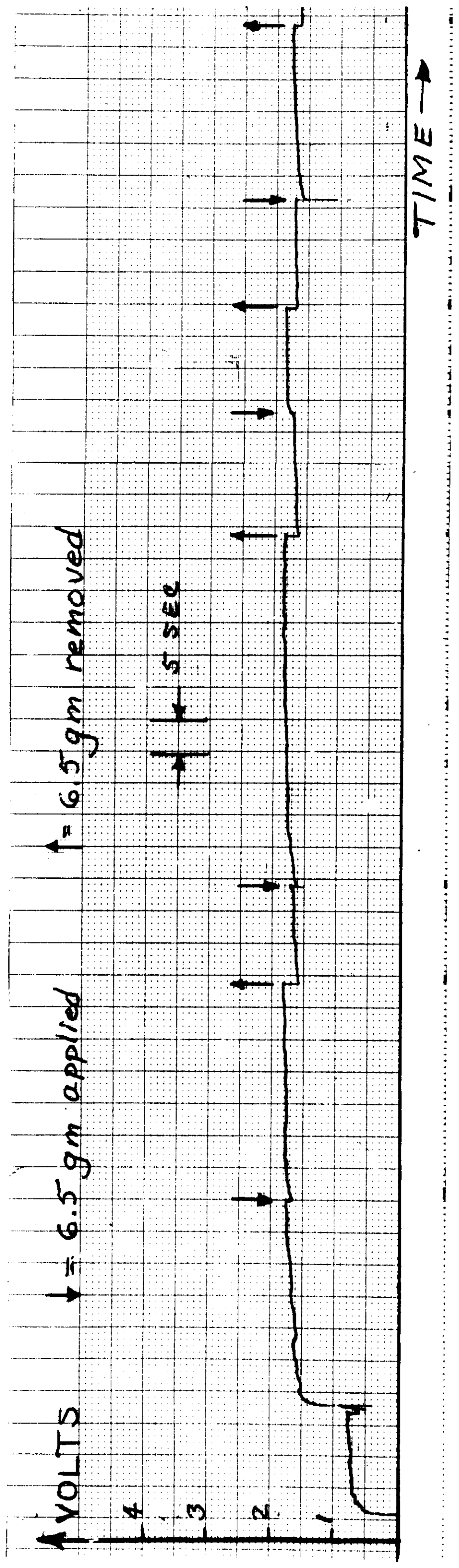
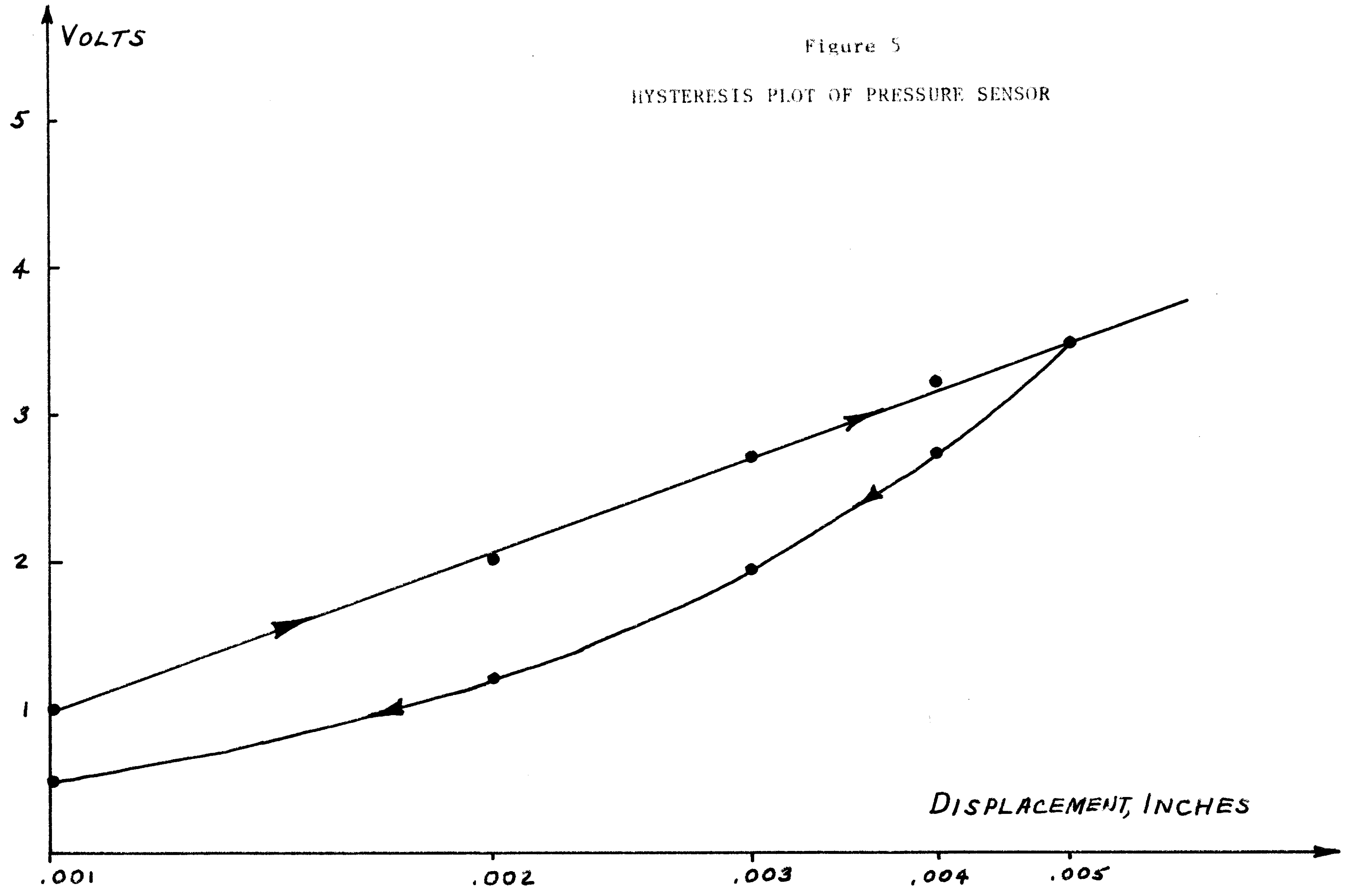


Figure 4. Sensitivity Test

Figure 5

HYSTERESIS PLOT OF PRESSURE SENSOR



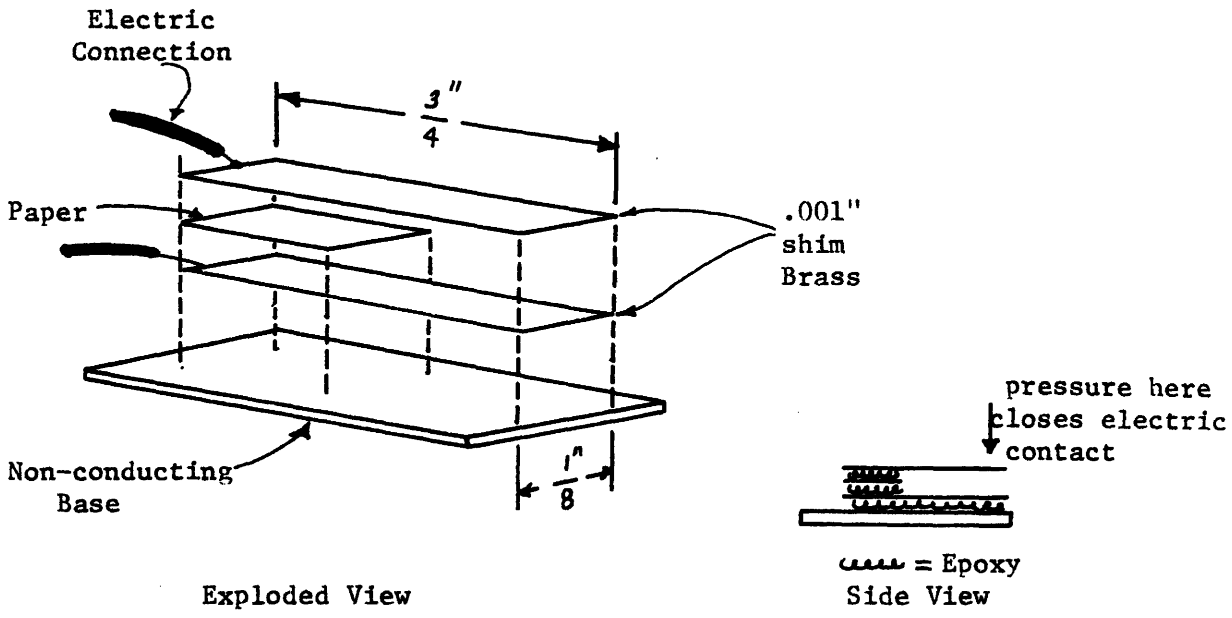


Figure 6

ON-OFF CONTACT SENSOR CONSTRUCTION

Contact with an object pushes the two pieces of brass together, completing an electric circuit. The brass is never stressed out of its elastic range, so no adjustments are needed due to prolonged use. Contact forces as low as 5 grams can be detected.

One of each type of sensor is shown, with a paper clip for size comparison, in Figure 7.

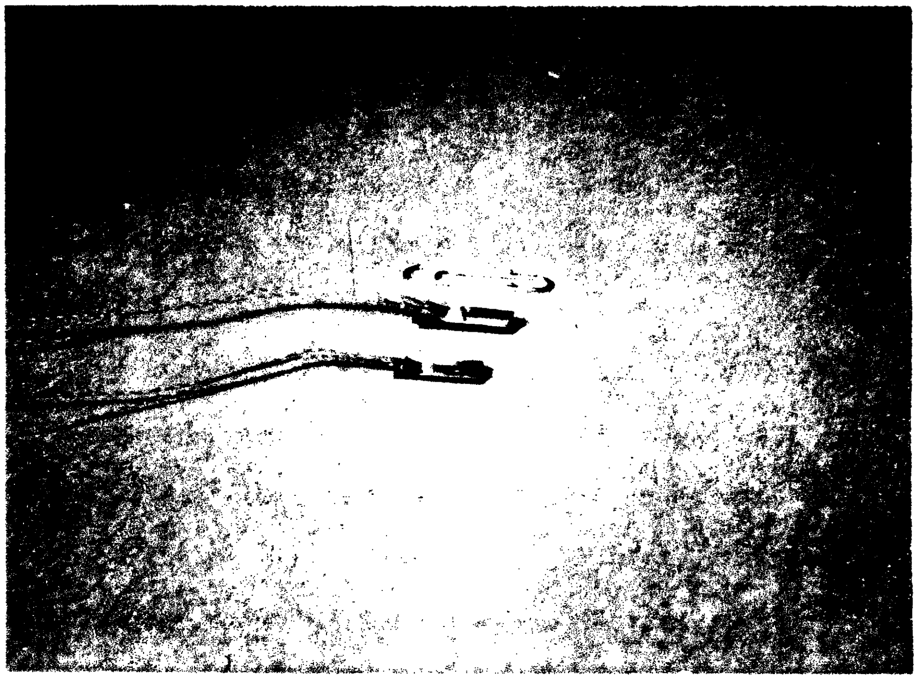


Figure 7

TOP TO BOTTOM: PAPER CLIP, ON-OFF CONTACT SENSOR, PRESSURE SENSOR

APPENDIX III

BRIEF DESCRIPTION OF DEMONSTRATION COMPUTER PROGRAM

This appendix describes briefly the computer program used in connection with the demonstration described in Chapter VI. After reading this appendix, one should be able to connect the apparatus to the computer and operate the apparatus using the program.

The program occupies registers 0 through 1777_8 and 3000_8 through 5177_8. The starting address is 5000_8. Table I is a storage map, showing the names of important routines and what memory page they are on. (See reference 8 for a discussion of memory pages.)

To set up the apparatus, one plugs the various Amphenol connectors into the appropriate receptacles on the back of the computer, as indicated by the tag on each connector. Next turn on the toggle switch labelled "Touch" on the apparatus, and the toggle switch on the motor translator. Load the program into core and we are ready. Figure 1 is a complete wiring diagram of the apparatus, showing how each motor is wired to its connector, how the touch sensors and limit switches are wired, and where each connector is to be attached to the computer.

The operator has access to the following teletype commands:

OPERATOR TYPES	WHAT HAPPENS
1. IS	1. Computer types INIT STATE=(Operator types (in decimal) a two digit x coord. and a two digit y coord. to show current location of jaws: 01,08 Computer types) Computer finds paths and types GR AT EQ. FS? READY (This means graph at equilibrium. Operator may type a final state.)
2. FS	2. Computer types FINAL STATE=(Operator types desired jaw coordinates. Computer types) If there is no path to this state, Computer types NO PATH READY Otherwise, computer types READY

3. GO (Note: The sequence IS, FS, GO is used to move the jaws around. IS is used to initialize the program.)	3. Jaws are moved from named initial state to named final state. If there is a collision, jaws will halt. Computer will give coords. of object collided with and demand a name. Operator may type a) NX (please ignore the object) b) W (it is a wall) c) any two characters not used for anything else (the name of a moveable object) Computer prepares a new path and motion continues until the final state is reached. Then it types READY
4. TP	4. Computer types out the path found by IS, PK, or CY, coded as directions for each step. 0 through 7 are compass directions for jaw motion, 8 and 9 are OPEN and CLOSE. TP is mainly diagnostic.
5. OP	5. Jaws open.
6. CL	6. Jaws close.

7. PK	7. Computer demands the name of object. Jaws then move to this object, if there is a path, and grasp it. Collisions are handled as in 3.
8. CY (Note: The sequence PK, CY is used to pick up an object, then carry it somewhere.)	8. Computer demands coords. of desired object location, then jaws carry object there, if there is a path. Collisions are handled as in 3.
9. H	9. Jaw motion stops.
10. RE	10. If used after H, while a path is being executed, motion on this path resumes.
11. KS	11. On-off contact sensors are deactivated.
12. RS	12. On-off contact sensors are re-activated.
13. NO	13. Operator may name an object. Computer demands object's coords. first, then the name, as in 3.
14. BL	14. The graph is cleaned of all objects and walls. The lists

	OBTABL and WALTBL are not disturbed. Diagnostic.
15. RT	15. Jaws retreat to last successfully occupied state. New path is found from there and is executed immediately if all Switch Register switches are down. Otherwise computer types READY
16. NR 17. SO 18. ES 19. WS	16, 17, 18, 19. Jaws move one quantization unit in the given direction (north, south, east, or west), then halt. When used with H, these commands enable the operator to register the jaws with quantization squares. Forcing the apparatus by hand with the motors turned off is not recommended! When operator uses these commands with OP and CL, he has manual control.

TABLE I

STORAGE MAP

Page (octal)	Important Routines
0	Interrupt service, misc. storage
1	Command interpreter (CMDINT)
2	Retrieval of path (GTPATH) and application of algorithm (ALGRTM)
3	Commands IS and FS
4	Commands GO and TP
5	Running the motors (CLOCK and SETMOT) and beginning of routines following collision. (SENSOR)
6 and 7	Routines following collision, to estimate object's location and to take in name supplied by operator
10, 11, 12, 13	Storage of graph
14	Routines for setting up the graph. The path is stored between 3300_8 and 3377_8 on this page.
15	Message typeout routine (MESAGE)
15 and 16	Messages
17	OBTABL, the list of names and locations of objects.

20	WALTBL, the list of wall locations
21	Commands PK and CY
22	Commands OP and CL, plus associated analog-digital conversion routines.
23	Manual control commands OP, CL, NR, SO, ES, WS.
24	Startup routine and on-off contact sensor receipt and interpretation

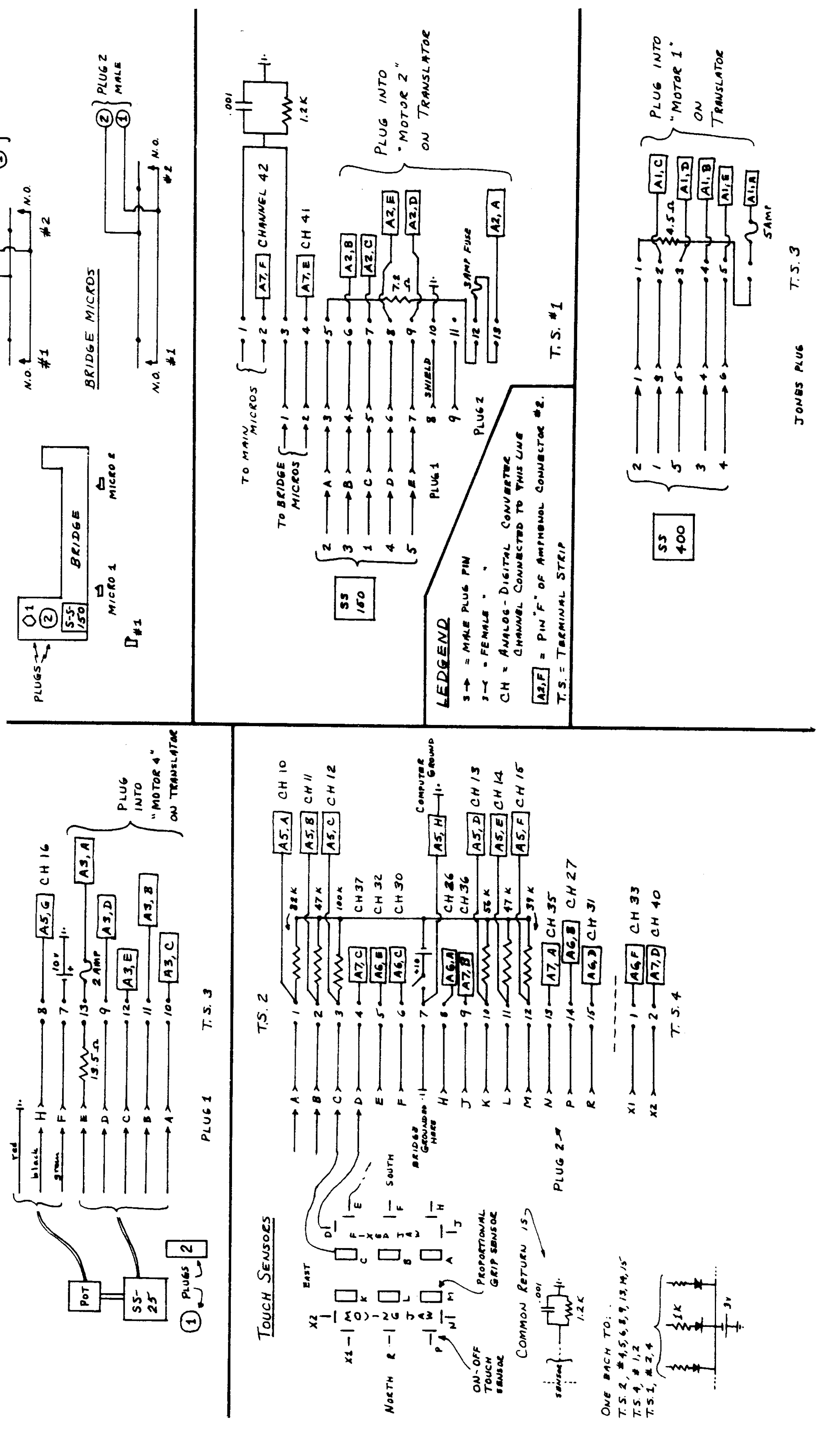


Figure 1

WIRING DIAGRAM FOR PLOTTING TABLE MANIPULATOR

BIOGRAPHICAL SKETCH

Daniel E. Whitney was born in Chicago, Illinois on June 8, 1938 and received his elementary education in the public schools of Winnetka, Illinois. He attended M.I.T. from 1956 to 1961, receiving Bachelor's degrees in Humanities and in Mechanical Engineering. He won the Boit Essay Prize during his junior year. Following two years of service in the U.S. Navy, Mr. Whitney returned to M.I.T., where he received the degree of Master of Science in Mechanical Engineering. During his graduate years he was both a Research Assistant and a Teaching Assistant. During summers he has worked for Abbott Laboratories, Bell Telephone Laboratories and the M.I.T. Instrumentation Laboratory. He is a member of Tau Beta Pi and Sigma Xi. His publications include

"Propagated Error Bounds for Numerical Solution of Transient Response," Proceedings IEEE (Letters), V. 54, #8, August 1966, p. 1084.

"Forced Response Evaluation by Matrix Exponential," Proceedings IEEE (Letters) V. 54, #8, August 1966, p. 1089.

"Propagation and Control of Roundoff Error in the Matrix Exponential Method," Proceedings IEEE (Letters) V. 54, #10, October 1966, p. 1483.

REFERENCES

1. Athans, M., and P. Falb,. Optimal Control, New York: McGraw-Hill, 1966.
2. Barber, D.J., "MANTRAN, A Symbolic Language for Supervisory Control of an Intelligent Remote Manipulator," S.M. Thesis, M.I.T., Department of Mechanical Engineering, May, 1967.
3. Bellman, R., "On a Routing Problem," Q. App. Math., 16, 1958, p. 87.
4. Bellman, R.E., and S.E. Dreyfus, Applied Dynamic Programming, Princeton: Princeton University Press, 1962.
5. Berge, Claude, Theory of Graphs and Its Applications, New York: John Wiley and Sons, 1962.
6. Bradley, W.E., "Telefactor Control of Space Operations," Astronautics and Aeronautics, May, 1967, pp 32-38.
7. Chen, Y.C., and O. Wing, "Some Properties of Cycle-free Directed Graphs," Journal of the Franklin Institute, v 281 (4), April 1966, p 293.
8. Digital Equipment Corp., Maynard, Mass., "PDP-8 User's Handbook."
9. Drake, A., Fundamentals of Applied Probability Theory, New York: McGraw-Hill Book Co., 1967.
10. Ernst, H.A., "A Computer-Operated Mechanical Hand," Sc.D. Thesis, M.I.T., Department of Electrical Engineering, December, 1961.
11. Ferrell, W.R., "Remote Manipulation with Transmission Delay," IEEE Trans. Human Factors in Electronics, HFE-6 (1), September 1965, pp 24-32.

12. Ferrell, W.R., "Delayed Force Feedback," Human Factors, Oct., 1966, pp 449-455.

13. Ford, L.R., Jr., "Network Flow Theory," Rand Corp. Paper P-923, August 14, 1956.

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

15. Greene, P.H., "New Problems in Adaptive Control," chap 18 of Computer and Information Sciences, ed. Wilcox, R.E., and J.T. Tom, Washington: Spartan Press, 1963.

16. Hart, P.E., N.J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," Stanford Research Institute unpublished memo, June, 1967.

17. Johnsen, Edwin, Discussant at 11th Annual Meeting of the Human Factors Society, Sept. 25-28, 1967.

18. Larson, R.E., "Dynamic Programming with Reduced Computational Requirements," Trans. IEEE Auto. Control, April, 1965, pp 135-43.

19. LISP 1.5 User's Manual, 2nd printing, M.I.T. Press, 1965.

20. McCandlish, S.G., "A Computer Simulation Experiment of Supervisory Control of Remote Manipulation," S.M. Thesis, M.I.T., Department of Mechanical Engineering, June, 1966.

21. "The Humanoids are Coming to Do The Dirty Work," Product Eng., 38 (17), Aug. 14, 1967, pp. 30-32.

22. Mergler, H.W., and P.W. Hammond, "A Path Optimization Scheme for a Numerically Controlled Remote Manipulator," 6th Ann. Symposium of the IEEE Human Factors in Electronics Group, May, 1965.

23. Minsky, M.L., "Steps Toward Artificial Intelligence," Proc. IRE, v 49, Jan. 1961, pp 1-30.

24. Minsky, M.L., and S.A. Papert, Research on Intelligent Automata, Status Report II, Sept., 1967. M.I.T. Project MAC.

25. Mooers, Calvin N., "TRAC, A Procedure-Describing Language for the Reactive Typewriter," Comm. A.C.M., (9),3, March, 1966, pp 215-219.

26. Mosher, R., "Dexterity and Agility Improvement," paper delivered to the 1965 Underwater Technology Meeting, ASME, New London, Conn.

27. Mosher, R., "Industrial Manipulators," Scientific American, October, 1964, p. 88.

28. 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, ed. Feigenbaum, E.A., and J. Feldman, New York: McGraw-Hill Book Co., 1963, pp 109-133.

29. Newell, Shaw and Simon, "Chess Playing Programs and the Problem of Complexity," in Feigenbaum and Feldman, op. cit., pp 39-66.

30. Newell, A., and H.A. Simon, "GPS, A Program That Simulates Human Thought," in Feigenbaum and Feldman, op. cit., pp 279-296.

31. Newell, A., and F.M. Tonge, "An Introduction to Information Processing Language IPL-V," Communications of the A.C.M., 3, 1960, pp 205-11.

32. Ore, Oystein, Theory of Graphs, Providence: American Math. Soc., 1962.

33. Polya, G., How to Solve It, Anchor Press, 1954.