

A.P. Ambler
H.G. Barrow
C.M. Brown
R.M. Burstall
R.J. Popplestone

Department of Machine Intelligence
University of Edinburgh

Abstract

A versatile assembly system, using TV cameras and computer-controlled arm and moving table, is described. It makes simple assemblies such as a peg and rings and a toy car. It separates parts from a heap, recognising them with an overhead camera, then assembles them by feel. It can be instructed to perform a new task with different parts by spending an hour showing it the parts and a day or two programming the assembly manipulations. A hierarchical description of parts, views, outlines etc. is used to construct models, and a structure matching algorithm is used in recognition.

1. Introduction

A computer-controlled versatile assembly system has been programmed during the past 12 months using the Edinburgh hand-eye hardware (Barrow and Crawford¹). The equipment (Fig. 1) consists of a moveable table, a mechanical hand with sensors and rotating palms, and two TV cameras, all connected via an 8K Honeywell 316 to a 128K time-shared ICL 4130 running POP-2 programs. Several other programs are running on this equipment, including a program for recognising irregular objects and one which packs arbitrarily shaped objects into a box (Michie et al²). The program described here is our most ambitious effort. It is capable of assembling a variety of structures, and much of our effort has been spent in enabling the machine to acquire descriptions of the parts for itself using an overhead TV camera.

Related work has been carried out at Hitachi (Ejiri et al³), at MIT (Winston^{4,5}) and at Stanford University (Feldman⁶). The Hitachi program could build a variety of simple structures of blocks from line drawings of the structure, the MIT programs can learn concepts about structures and copy an arbitrary structure of simple blocks given spare parts, and a recent Stanford program can assemble a simple automobile water pump using preprogrammed hand manipulations.

2. The Task

A number of parts are placed by the operator in a heap on the table (Fig. 2, peg and rings). The machine's task is to separate the parts and recognise them, then to assemble them into some predetermined configuration (Fig. 3). Figs. 4 and 5 show another example, a toy car. We are currently thinking in terms of up to a dozen parts with outlines described by up to twenty or so straight or curved segments from any one view, possibly with some holes of similar complexity.

In order to explore the capabilities of a computer-controlled system as opposed to a conventional electromechanical device we seek a versatile assembly system. The demand for versatility is also calculated to raise interesting aspects from an Artificial Intelligence point of view.

Our goal has been to develop a system which enables one to:-

- (i) think up a new assembly involving a kit of parts which have not been used before,
- (ii) spend a day or so familiarising the machine with the appearance and manipulation of the parts and instructing it how to assemble them into the required structure,
- (iii) leave the machine unattended, busily making the structures ad nauseam, provided that a fresh heap of parts is dumped on the table from time to time.

We have achieved this goal for simple structures. We can show the machine half a dozen new parts and instruct it how to lay them out ready for assembly in about two hours; interactively programming the assembly operations themselves takes four hours or so. The machine can make the structures like the peg and rings, the toy car or a toy ship unassisted, but slowly, taking an hour or two to find and assemble the parts. It completes the assembly correctly about four times out of five. It should be possible to ease the programming of the assembly operations somewhat, but removing the need to program movements in terms of numerical co-ordinates would need a new approach. The system is about 50K of POP-2 code.

3. How The System Performs the Task

The system transforms a heap of parts into a completed assembly, typically going through the following steps:-

Layout: Identify parts visually and put them in standard positions.

1. Locate all parts or heaps of parts on the table using the side camera (wide angle).
2. Inspect each part or heap with the overhead camera. If it is recognisable as an isolated part pick it up and lay it out in a standard position and orientation.
3. If there are no parts or heaps left, or if all the required parts have been found go to step 5.
4. Using the overhead camera look for a protrusion in the smallest heap, grasp the protruding part or parts and separate them from the heap and go to step 1. If it has no protrusions try to break up the heap by picking it up as a whole and turning it over so that it falls apart. If this does not work, try pushing it with the hand at various heights. If still no success, try another heap. Go to step 1.
5. If some parts are missing, complain. If there are more parts than are needed, clear them away.

Assembly:

Pick up each part from its standard position and insert it into the assembly, by feel. A workbench with a simple vice and various working surfaces is used.

The program is written as two quite distinct sub-programs, layout and assembly. At present they do not communicate; they both know the standard

positions and orientations of the parts. The layout subprogram uses descriptions of the parts, acquired during the instruction phase, to recognise them, pick them up, turn them over if necessary and put them into standard positions. The assembly subprogram works blind, using no internal descriptions of the parts. It is written interactively during the instruction phase.

The system carries out the steps listed above at 'execution time'. It is able to do so because the following steps have been carried out previously at 'instruction time'.

A. Instructing the layout program

For each part in each stable state (e.g. right way up, or on its side).

1. The operator places the part on the table under the vertical camera, the machine takes a picture and creates from it an internal description of that view of the part. This is repeated several times, and the machine adjusts its descriptions each time, taking note of the variations caused by reorientation of the part and imperfect picture information.

2. The programmer types in on-line commands to move the hand, to pick up the object, turn it over, put it down and pick it up again if necessary, then to put it down in the standard position and orientation. (If the assembly has several identical parts separate commands are given for putting each one down in its own place.) The programmer intersperses these commands with instructions to remember the current state, e.g. when the hand has closed over the part. The system takes a note of these specified states and at execute time constructs a sequence of actions to put the part into its standard position and orientation.

B. Writing the assembly program

For each part

1. The programmer puts the part on the table in its standard position and orientation, and he interactively devises and edits some POP-2 program to make the machine pick up the part and fit it into the assembly. His program uses basic move and grasp operations, and two high level manipulation operations provided for constrained moves and hole fitting.

4. The Layout Subprogram: Descriptions

We must first explain the kind of internal descriptions of parts used in the layout program. We can then show how it creates these descriptions at instruction time and how it uses them to recognise parts at execution time. For clarity we have simplified a few programming details, glossing over some unnecessary or uninteresting distinctions.

The program works in terms of a hierarchy of concepts called entities, each represented by program data structures, having other entities as its components. The entities used are summarised in Table 1 and Table 2 gives a brief description of each.

Table 1

Hierarchical structure of entities

An entity is either a table top, or an object-set, or an object, or a part, or a hand, or a workbench, or a heap, or a stable state, or a view, or a region, or a hole-set, or a hole, or an outline, or a segment.

A table top has an object-set.

An object-set has objects.

An object is either a part, or a hand, or a workbench, or a heap.

A part has stable states.

A stable state has a view.

A view has a region.

A region has an outline and a hole-set.

A hole-set has holes.

A hole has an outline.

An outline has segments.

Table 2

The entities used

The Table top is the whole collection of things on the table.

An Object is any physical thing on the table which can be seen or touched; it is initially distinguished from its surroundings by clear space on the table.

A Part is one of the separate pieces needed for the assembly e.g. one of the wheels of the car.

A Stable-state is one of the states in which a part can rest on the table, irrespective of orientation or position e.g. on its side, upside down. There should be only a small number of distinguishable such states.

A View is an analysed TV picture.

A Region is a connected light area in a picture, possibly with darker holes (we use light objects on a dark background).

A Hole is a dark area inside a region.

An Outline is the outer boundary of a region or hole.

A Segment is a segment of a circle (up to 360°) with specific length and curvature (zero curvature means a straight segment). The irregular boundary of a region or hole is analysed into a small number of segments by curve fitting (Fig. 6).

Each entity either has an n-tuple of components, or it has a set of components. The size of the n-tuple is fixed as in the above 'syntax', for example a region has a pair of components; but the size of the set is not fixed until instruct time, for example the system discovers that the hole-set of a car body side view has two holes.

An entity may possess properties and some relations (at present only binary ones) may subsist between its components. The properties and relations have names and may be truth-valued or take values in some other domain such as numbers.

We make an important distinction between two kinds of entities: model and individual. Each has entities of its own kind as components. An individual entity is an internal description generated by a particular exposure to a physical object using information from TV camera and hand sensors. Thus when the operator puts a car body on the table, the machine takes a picture, turns it over and takes another picture, one individual entity of type 'part' is generated and two individual entities of type 'view' are generated, together with individual outlines, holes, segments etc. A model entity, on the other hand, is a summary or composite of a number of such experiences. The end result of the instruction phase is a collection of model entities incorporating the system's knowledge about the parts, their views, outlines etc.; the individual parts, view and out-

lines which gave rise to these will have been discarded.

The important operation in recognition is the creation, from visual and tactile sense data, of an individual entity which matches a model entity and whose components match the components of the model entity. The individual entity contains a pointer to its model and one to its sense data, thus binding them together; it also contains certain specific information, e.g. position and orientation, not appropriate to model entities.

There are a few exceptions to the above. The system does not create models for its hand or the workbench at instruction time; these are given beforehand. There is no model for a heap since an individual heap is generated by elimination, on failure to recognise a part, hand or workbench.

The system has a data structure for each model entity and individual entity; these are POP-2 records linked by pointers to their components into tree structures. There is also a data structure for each entity class, for example the class 'view' and the class 'region'. Each model or individual entity belongs to some class and certain data pertains to the class as a whole - for example a list of the properties which an entity of that class enjoys and functions for computing their values.

To build entities the machine needs raw material which we shall call 'sense data', information from the TV camera or possibly the hand sensors which has not yet been recognised as referring to any known entity. The recognition process, which we describe below, takes a model entity and some sense data and tries to create an individual entity which corresponds to the model.

Table 3 summarises the four notions of entity class, model entity, individual entity and sense data showing what information is associated with each. The use of this information will be clearer when we discuss the recognition process.

The matching process which recognises parts

To understand the recognition process let us consider what happens when the system has taken a TV picture and tries to interpret it as a side view of a car body. This may be during the instruction phase when it has been told that it is looking at the side view of a car body, or during the execution phase after it had found an upright car body and turned it over. Or again it might be dealing with an unknown object, and 'sideview of car body' might be just one possible interpretation among several which it was trying.

The TV picture, a 2-dimensional array of brightness levels, constitutes a sense datum, d in D . A matching function is now applied to this sense datum and the model of the side view of the car body. This function produces a set of individual side views of car bodies, an empty set if there is no way of interpreting the picture as such a view, otherwise one element for each possible interpretation. Thus

match: Sense data $\times$ Models $\rightarrow$ Set of individuals

The matching function works its way recursively down the hierarchy from top to bottom, comparing gross properties on the way down and failing if they are too discrepant. Thus it might fail because the area of the region it is looking at in the picture is too small for a car body, without bothering to analyse the outline of the region. As it goes down it refines the sense data, using a thresholding region finder routine to get region level sense data from the view level brightness array, finding holes with the same routine to get hole data and fitting

Table 3

Notion and symbol	Associated descriptive information
Entity classes, C	Property names P , Relation names R , Property finding functions $f: D \rightarrow V$ for $p \in P$ (V is the set of values for properties and relations), Relation finding functions $f: D \rightarrow V$ for $r \in R$, Classes of components, Component finding functions in $D \rightarrow D$, matching function in $D \times M \rightarrow 2^1$.
Model entities, M	Class, values of properties and relations, components.
Individual entities, I	Class, values of properties and relations, components, parameters, pointers to model and to sense data.
Sense data, D	At view level: 2-D brightness array from TV picture. At region level: region perimeter as list of vector increments, 2-D Boolean array showing whether point is inside and brightness array. At hole level: as for region level referring to dark holes At segment level: segment length, curvature and position.

curves to the perimeters to get segment data. (We have called the region finder and curve fitter 'component finding functions'). When it gets to the bottom level the recursion unwinds and passes up the hierarchy descriptions of individual entities, using their finer properties and relations between them to establish correspondence with the model. At each level the matching function produces a set of individual entities each of which might correspond to the model, thus dealing with ambiguity essentially as would a 'back-track' or nondeterministic process.

To be more precise, the function 'match' works as follows:-

Function match(d, m)

Let c be the entity class of m .

Let f be the special matching function of class c .

result = $f(d, m)$

The special matching function f may vary from class to class, but normally f is 'general-match', defined as follows:-

General-match: Sense data $\times$ Models $\rightarrow$ Sets of individuals

Function general-match(d, m)

Let c be the entity class of m .

Let P be the set of properties for class c .

For each p in P , compare $f(d)$, the value of property d for the sense datum, with the value of p in the model m . If there is too much discrepancy exit with result = empty set.

Case 1. m has an n -tuple of components, $m_1, \dots, m_k$.

Apply the component finding functions of the class c to d , to find sense data relevant to the components say $d_1, \dots, d_n$. As each d_j is computed, match it against the component m_j , thus let $I_j = \text{match}(d_j, m_j)$. If some I_j is empty then exit immediately with result = empty set. Otherwise use each element of $\prod_{j=1}^n I_j$ (the Cartesian product of the sets of individual components) to construct a new individual with these components. The result is the set of

these individuals.

Case 2. m has a set of components, S_m . Apply the component finding functions^m for class c to find a set of sense data relevant to the components, S_c . Use the relation value finding functions of $\bar{c}$, f_r for r in R , to compute the values of relations between the S_c . Compare these with the known values of the relations for the S_m and use the relational structure matching algorithm (described below), together with the function match, to find the largest subsets of S_c which correspond. If these subsets are sufficiently large construct a new individual entity from each correspondence among the components. The result is the set of individuals so constructed.

5. Matching Relational Structures

A set of segments forming the outline of a part can be regarded as a relational structure endowed with properties, such as length and curvature, and relations, such as adjacency, distance or relative orientation, similarly for holes or objects on the table top. Although the properties and relations usually take numerical values (length) it will simplify the discussion to talk in terms of truth valued ones (long, medium, short). In the algorithm given in the last section there is a point (case 2) where we need to put two sets (of segments, say) into correspondence on the basis of these properties and relations. TV picture processing being what it is we expect discrepancies (segments missing, two segments coalesced) but we want to match as many elements as possible. Fig. 7 shows two simple outlines and corresponding relational structures; they have several common substructures, e.g. {11', 32', 54', 43'}.

More precisely, by a relational structure we mean a set S of elements together with a set of properties P and a set of relations R over it (we consider only binary relations here). Given two relational structures $\langle S_1, P, R \rangle$ and $\langle S_2, P, R \rangle$ we define a match between them as a set $T_1 \subseteq S_1$, a set $T_2 \subseteq S_2$ and an isomorphism, $\approx$, between T_1 and T_2 preserving properties and relations. Thus $s_1 \approx s_2$ implies $p(s_1)$ iff $p(s_2)$ for each p in P , also $s_1 \approx s_2$ and $s_1 \approx s_3$ imply $r(s_1, s_3)$ iff $r(s_2, s_3)$ for each r in R , and a match represents a common substructure in our two relational structures.

We can find matches as follows. By an assignment we mean a pair $\langle s_1, s_2 \rangle$ with s_1 in S_1 and s_2 in S_2 such that $p(s_1)$ iff $p(s_2)$ for each p in P . In Fig. 7 the assignments are: 11', 12', 14', 23', 31', 32', 34', etc. We say two assignments $\langle s_1, s_2 \rangle$ and $\langle s_1', s_2' \rangle$ are compatible if $r(s_1, s_1')$ iff $r(s_2, s_2')$ for all r in R . Now by definition a match is just a set of assignments such that each assignment is compatible with every other assignment in the set. Indeed we may think of the assignments as forming the nodes of a graph with compatibility as the (symmetric) relation forming the arcs. Our problem then is to find totally connected subsets of this graph, often called cliques.

A clique is said to be maximal if no other clique properly includes it. Finding maximal cliques is a well known problem (Karp⁷). A graph of n assignments may have $(n/2)^{n/2}$ maximal cliques in a theoretical bad case, but at least we can find each maximal clique in time proportional to n^3 (or $n \log n$ if we push it). We can do this by using a refinement of a simple binary search algorithm given by Burstall⁸. Knödel⁹ gives a similar algorithm. In fact for our recognition problem it seems adequate to generate only largest

cliques (those with maximal number of elements) rather than maximal ones. Fig. 8 shows the compatibility graph for our previous example and the three maximal cliques indicated by Δ , $\square$ and $\circ$. (Remember nodes in this graph represent corresponding pairs of elements of the two structures and connecting arcs represent compatibility between such pairs.) In practice we do not seem to get too many largest cliques when dealing with relational structures of a dozen or so elements, and the algorithm takes only a few seconds or so to generate them.

Our clique finding algorithm builds up cliques a node at a time. It is a recursive function, cliques: $2^{Nodes} \times 2^{Nodes} \rightarrow 2^{Nodes}$ such that cliques(X, Y) is the set of all cliques which include a clique X and are included in Y . Thus cliques($\emptyset, Nodes$) is the set of all cliques in the graph. It is defined most simply thus

$$\text{cliques}(X, Y) \leftarrow \begin{cases} \text{if no node in } Y-X \text{ is connected to all} \\ \text{elements of } X \text{ then } \{X\} \\ \text{else } \text{cliques}(X \cup \{y\}, Y) \cup \text{cliques}(X, Y-\{y\}) \\ \text{where } y \text{ is such a node.} \end{cases}$$

(The reasoning is that if some y is eligible for addition to the clique X then each clique including X must either contain y or exclude y .) This generates a search tree whose nodes are labelled with a pair: set of nodes chosen, set of nodes available for choice. It starts with the empty set and all the nodes and each branch consists of including or excluding a node.

If we are looking for maximal cliques only we can cut down the search by computing at each state the set $\bar{Y}$ where $\bar{Y} = \{z \text{ in } Nodes: z \text{ is connected to each node in } Y\}$ and noting that any maximal clique includes $\bar{Y}$, also noting that if $\bar{Y} \not\subseteq Y$ then there are no maximal cliques included in Y . In fact our program seeks only largest cliques and runs a version of the function cliques of size k , stopping the recursion if the size of X plus the number of nodes in $X-\bar{Y}$ connected to all of X becomes less than k . We count down on k until some cliques are found.

In this section we have defined an assignment as a pair of elements, one from each structure, which have identical property values. More loosely one can demand only some degree of similarity in the properties. Our program uses as the assignments the set of all pairs $\langle m', i' \rangle$ such that i' is in match(d', m') for some m' in S_m and some d' in S_d . This gives us the nodes in our graph; the arcs are found by defining compatibility to be some suitable degree of similarity in the relations.

This relational structure matching is used three times by the layout program: to match segments in an outline, to match holes in a region and to match objects on a table top, e.g. associate some objects which it now sees with its previous knowledge. It gives us a robust matching technique which can make correspondences between observed and predicted elements in spite of imperfect data. It has some similarity in its way of working to Waltz's technique for labelling pictures (Waltz¹⁰). Winston also does structure matching. Graph isomorphism is a well-known problem (e.g. Corneil and Gottlieb¹¹); relational structure isomorphism is essentially the same. Testing whether one structure is a substructure of another is computationally harder (Barrow, Ambler and Burstall¹²) and our problem here, finding common substructures is still more onerous, but it is needed if both structures have imperfections.

Separating parts from a heap

The process of removing parts from a heap is heuristic, that is it needs a little luck. But it has never failed, if the machine is left to worry at a

heap long enough. Our program does not detect internal lines, relying on the outline of the heap. It attacks the smallest heap first.

The first tactic used is to look for a protrusion in the heap outline with a 'neck' which might indicate a part which is easily separable. A number of possible hand positions are then considered so as to pick up the protruding part or parts without fouling the rest of the heap. If it succeeds in picking something up the machine examines both this and the rest of the heap, trying to recognise an isolated part. The second, cruder, tactic is to try and pick up the whole heap, then rotate the palms to let pieces fall off. The third is to push the heap with the hand at various heights. Clearly there are subtler ways of picking objects out of a heap, but for the time being our simple tactics suffice.

Overall control aspects of the layout program

The control structure of the layout program is quite simple. At the top level there is a simple loop. We may transcribe the POP-2 code thus

```
loop:  if all-tidy then goto exit;
      if do-search then explore (work-area)
      else if parts-untidy then put-all-away
      else if extra-objects then discard-objects
      else if lost-parts then search-for-missing-
                                   parts
      else if parts-needed then smash-a-heap
      else if extra-heaps then put-heaps-in-
                                   corner; goto loop
exit:
```

This enables it to cope with too many or too few parts. If one of the lower level routines encounters difficulties it simply jumps out to loop. At a lower level various routines on entry assign a label to the variable pick-up-fail, and failure to pick something up satisfactorily causes a jump out to the current label.

Our recognition process is basically recursive (we have experimented with some process-swapping techniques but they are not in our program at present). This does not lead to too much rigidity, because we make rather extensive use of 'memo-functions' (Michie¹⁴); for example, when an analysis of the outline of a region is required the analysing function remembers the result and, when asked for it again, simply returns it immediately. Thus outline analysis is only done when needed and never repeated.

6. The Assembly Subprogram

This program works blind, using only hand sensing. It is written interactively at instruction time in terms of basic hand moving and sensing operations together with two higher level operations.

The basic operations include 'raise z centimetres', 'move to (x,y)', 'grasp to w centimetres', 'rotate palm by θ ', and functions for reading forces on the hand by means of strain gauges, namely gripping force, weight of the object held and torque. There are two higher level operations: constrained move and hole fitting.

Constrained move. This operation has two parameters, both force vectors, f_m a force opposing movement and f_c a constraining force (Fig. 9). Let u_m and u_c be the unit vectors in these two directions. Let ϵ_m and ϵ_c be small scalar distances. The hand attempts to move in direction $-u_m$ until it is opposed by a force larger than f_m . At the same time it keeps in contact with a surface which offers a resisting force f_c . The resulting movement will not necessarily be in direction $-u_m$ but along the surface

according to the component of $-u_m$ tangential to the surface. The operation works in detail as follows:-

1. If a force greater than f_m is felt then stop.
2. Move by $-\epsilon_m u_m$.
3. If a force greater than f_c is felt then move by $\epsilon_c u_c$ and goto 3, otherwise goto 1.

Fig. 9 shows an example of the pattern of movement which a single call on the constrained move operation might produce. The constraining force parameter may be left undefined giving an unconstrained move in direction $-u_m$ until an opposing force is felt. This operation was suggested to us by work at MIT Draper Laboratory (Nevins et al¹⁵).

Hole fitting. This operation has two force vector parameters f_m and f_s . Let u_m and u_s be the corresponding unit vectors and ϵ_c and ϵ_s small scalar distances. It is used when the hand is holding an object with a protrusion which must be fitted into a hole (e.g. axle into car body) or is holding an object with a hole which is to be fitted over some protrusion (e.g. putting a ring on the peg). The hand moves in a spiral pattern normal to u_c , pushing repeatedly in the direction u_c until an opposing force greater than f_c is felt and then retreating and making a short move in the direction of the spiral before trying again. When, at some point in the spiral it succeeds in moving forward by $\epsilon_c u_c$, it tests whether it has found the hole by moving sideways by $-\epsilon_s u_s$. If a force greater than f_s is encountered (or after five attempts) the process stops, otherwise the spiral pattern is resumed.

The wooden car assembly gives an idea of how the program proceeds. A 'workbench' is used, fixed to one corner of the table (Fig. 10). It has a 'vice' for holding a wheel while an axle is being inserted, consisting of an L-shaped corner piece and a pivoted bar which the hand closes so that the wheel is held between the bar and the L. It also has a vertical 'wall' so that the car body can be held firmly while the second wheel is pushed onto each axle.

The sequence of events, in outline is:-

- (i) The hand puts a wheel in the vice and inserts an axle.
- (ii) It turns the car body upside down, picks up the axle with the wheel on it and inserts it into the body.
- (iii) Repeat (i) and (ii) for the second wheel and axle.
- (iv) Put the car body against the wall upside down with the two wheels against the wall.
- (v) Push the remaining two wheels onto the protruding axles.
- (vi) Pick up the assembled car and place it on the table.

Assembly programming is still quite tedious, involving choice of numerical parameters for distances and forces, and we have some ideas for easing it, Popplestone¹⁶. There is clearly a lot of thinking to be done before we could make the assembly phase as versatile and easily instructable as the layout, e.g. by replacing numerical commands with instructions using relations like 'on top of' and 'fitting into' or by showing the machine intermediate assemblies. In particular our present assembly subprogram does not use the internal descriptions of the parts which have been acquired during instruction by the layout program. Such descriptions would have to be recast so as to be useful for assembly as well as recognition.

7. Concluding Remarks

Writing this program has been a valuable exercise

from the point of view of understanding what problems are important for an integrated assembly system. By tackling a definite task but imposing the requirement of versatility, we have raised some interesting Artificial Intelligence questions without writing a program just to justify A.I. dogmas. We hope to work in future on (i) making the learning system more coherent (ii) extending the vision system to deal better with side views (essentially requiring 3-dimensional techniques) and to analyse half-completed assemblies, and (iii) making it easier to program assembly manipulations, being less specific about numerical co-ordinates and the magnitudes of forces.

Acknowledgements

Donald Michie initiated our work on robotics and directed the efforts to produce a working hand-eye system. Steve Salter and Gegan Crawford made the main contributions to the hardware. Dan Bobrow made helpful suggestions about the vision and recognition side, also about this paper. Ken Turner contributed the outline segmentation routines to our vision work. Valuable ideas about assembly came from MIT Draper Labs. We are grateful to Eleanor Kerse for typing. The project had support from the Science Research Council and from the General Post Office.

References

- (1) Barrow, H.G. and Crawford, C.F. The Mark 1.5 Edinburgh Robot Facility. Machine Intelligence 7 (eds. B. Meltzer and D. Michie) Edinburgh: Edinburgh University Press, pp. 465-480. (1972).
- (2) Michie, D., Ambler, A.P., Barrow, H.G., Burstall, R.M., Popplestone, R.J. and Turner, K. Vision and Manipulation as a Programming Problem. Proceedings of the 1st Conference on Industrial Robot Technology, Nottingham. (1973).
- (3) Ejiri, M., Uno, T., Yoda, H., Goto, T. and Takeyasu, K. An intelligent robot with cognition and decision-making ability. Proceedings of Second International Joint Conference on Artificial Intelligence, Imperial College, London, pp. 350-358. (1971).
- (4) Winston, P.H. Wandering About on the Top of the Robot. Vision Flash 15, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts. (1971).
- (5) Winston, P.H. The MIT Robot. Machine Intelligence 7 (eds. B. Meltzer and D. Michie) Edinburgh: Edinburgh University Press, pp. 431-463. (1972).
- (6) Feldman, J.A. Private Communication. (1973).
- (7) Karp, R.M. Reducibility Among Combinatorial Problems. Complexity of Computer Computations (ed. R.E. Miller and J.W. Thatcher) New York: Plenum Press, pp. 85-103. (1972).
- (8) Burstall, R.M. Tree-searching methods with an application to a network design problem. Machine Intelligence 1 (eds. N.L. Collins and D. Michie) Edinburgh: Oliver and Boyd, pp. 65-85. (1967).
- (9) Knödel, W. Bestimmung aller maximalen vollständigen Teilgraphen eines Graphen G nach Stoffers. Computing, Vol. 3, No. 3, pp. 239-240. (1968). Correction in Ibid. 4, p. 75.
- (10) Waltz, D.L. Generating Semantic Descriptions for Drawings of Scenes with Shadows. Artificial Intelligence Laboratories Report AI TR-271, MIT, Cambridge, Mass. (1972).
- (11) Winston, P.H. Learning Structural Descriptions from Examples. Artificial Intelligence Technical

Report 231, Artificial Intelligence Laboratory, MIT, Cambridge, Mass. (1970).

(12) Cornell, D.G. and Gottlieb, C.C. An efficient algorithm for graph isomorphism. J. Assoc. comput. Mach., 17, 51-64. (1970).

(13) Barrow, H.G., Ambler, A.P. and Burstall, R.M. Some Techniques for Recognizing Structures in Pictures. Frontiers of Pattern Recognition (ed. S. Watanabe) New York: Academic Press, pp. 1-29. (1972).

(14) Michie, D. 'Memo' functions and machine learning. Nature, 218, 19-22. (1968).

(15) Nevins, J.L., Sheridan, T.B., Whitney, D.E. and Woodin, A.E. The Multi-Moded Remote Manipulator System. E-2720. Charles Stark Draper Laboratory, MIT, Cambridge, Massachusetts. (1972).

(16) Popplestone, R.J. Solving equations involving rotations. Research Memorandum MIP-R-99. Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh. (1973)

Reference on cliques which came to hand after paper was typed:-

(17) Akkoyunlu, E.A. The Enumeration of Maximal Cliques of Large Graphs. SIAM Journal on Computing, Vol. 2, No. 1, March. (1973).

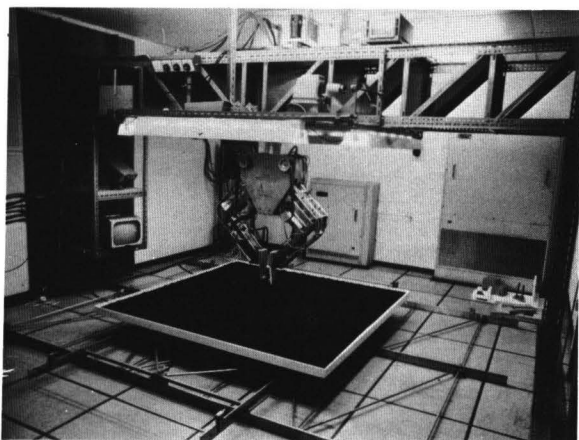


FIGURE 1



FIGURE 2

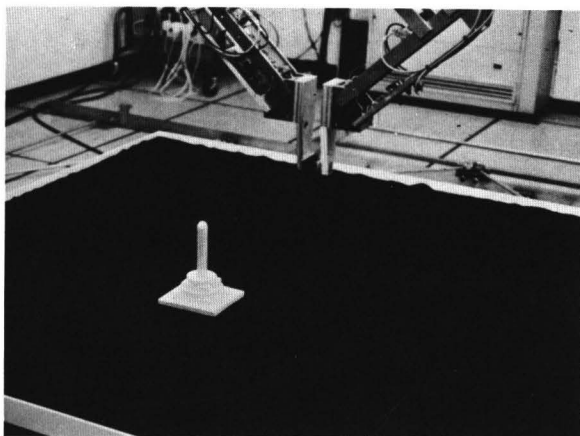


FIGURE 3

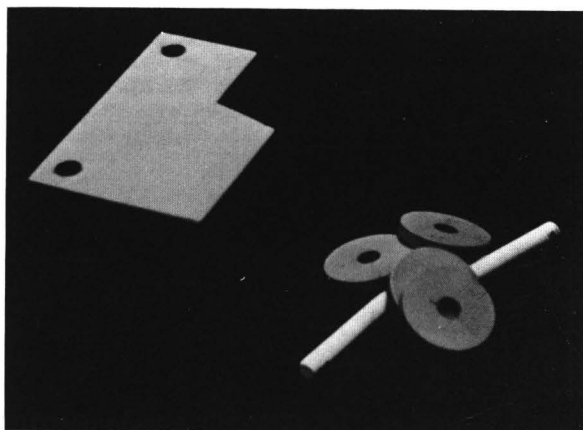


FIGURE 4

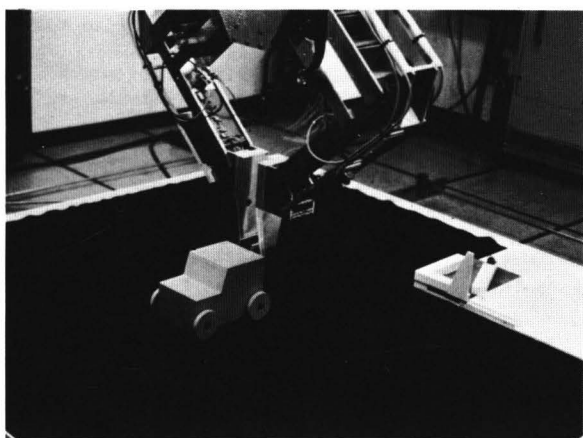
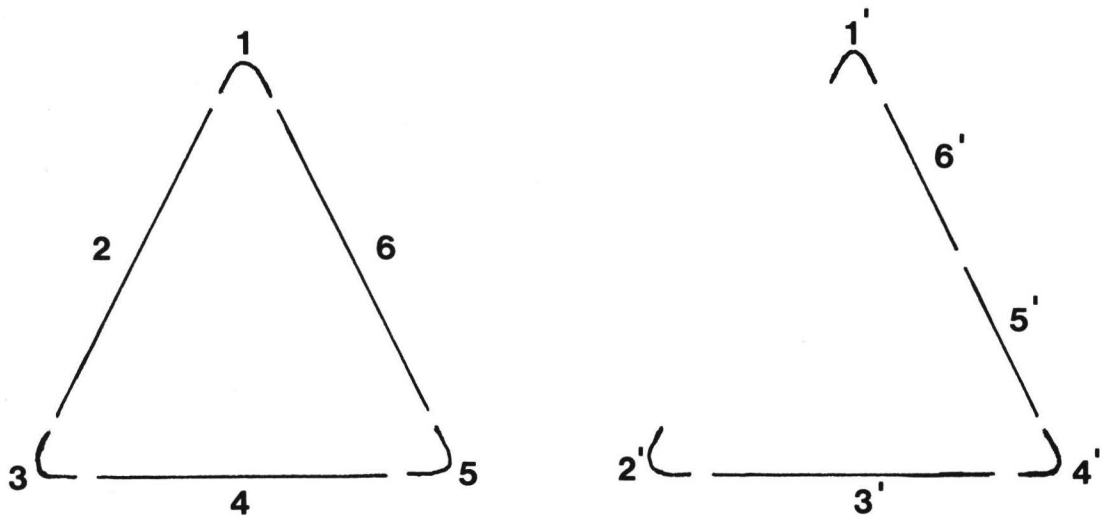


FIGURE 5



FIGURE 6

OUTLINES



RELATIONAL STRUCTURES

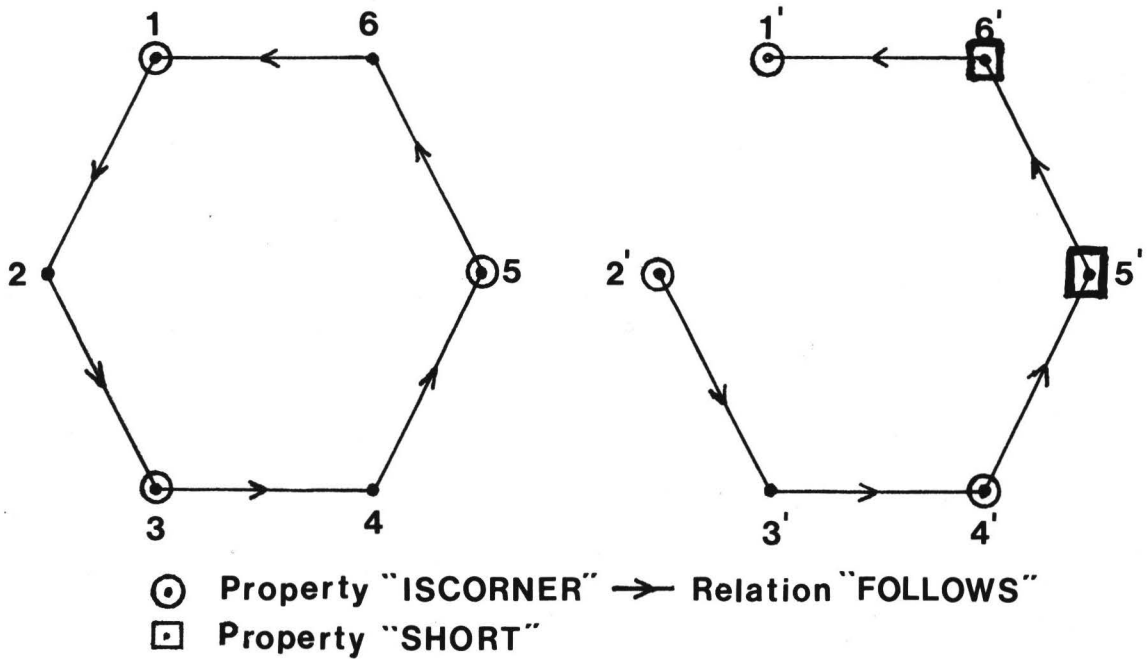


FIGURE 7

MAXIMAL CLIQUES

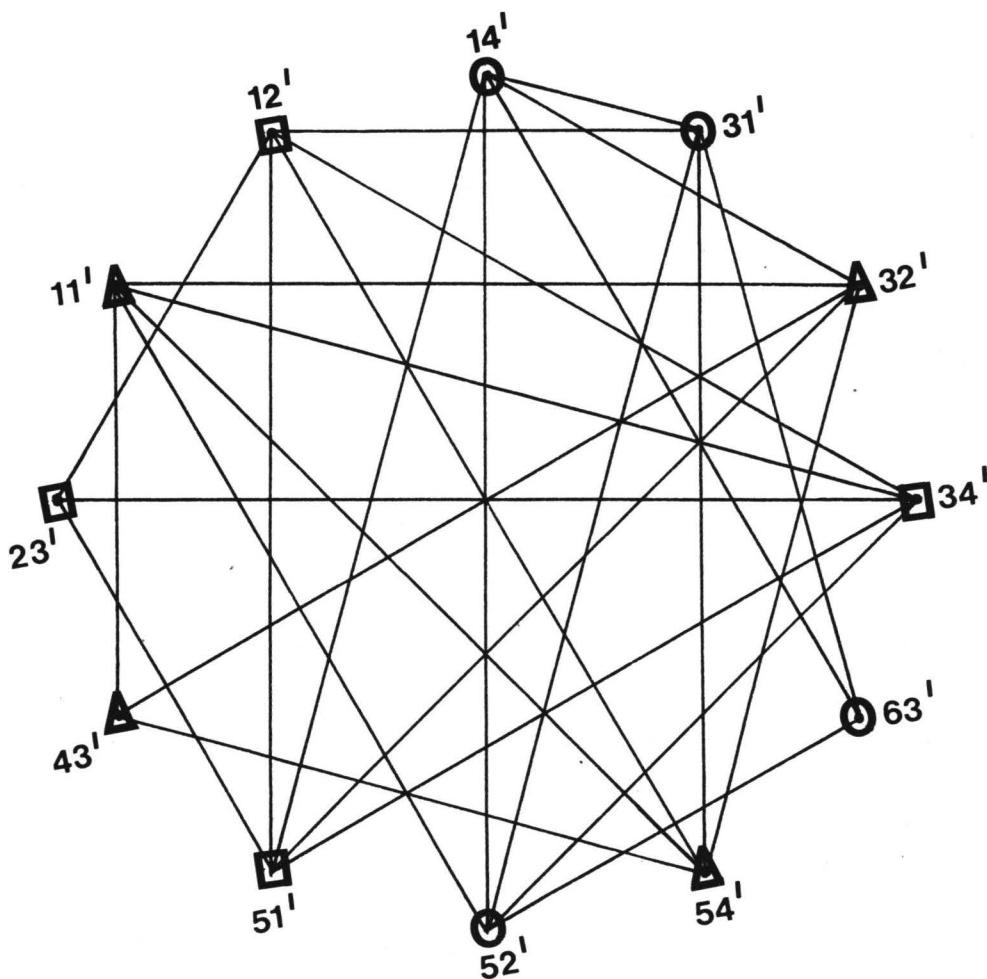


FIGURE 8

A CONSTRAINED MOVE

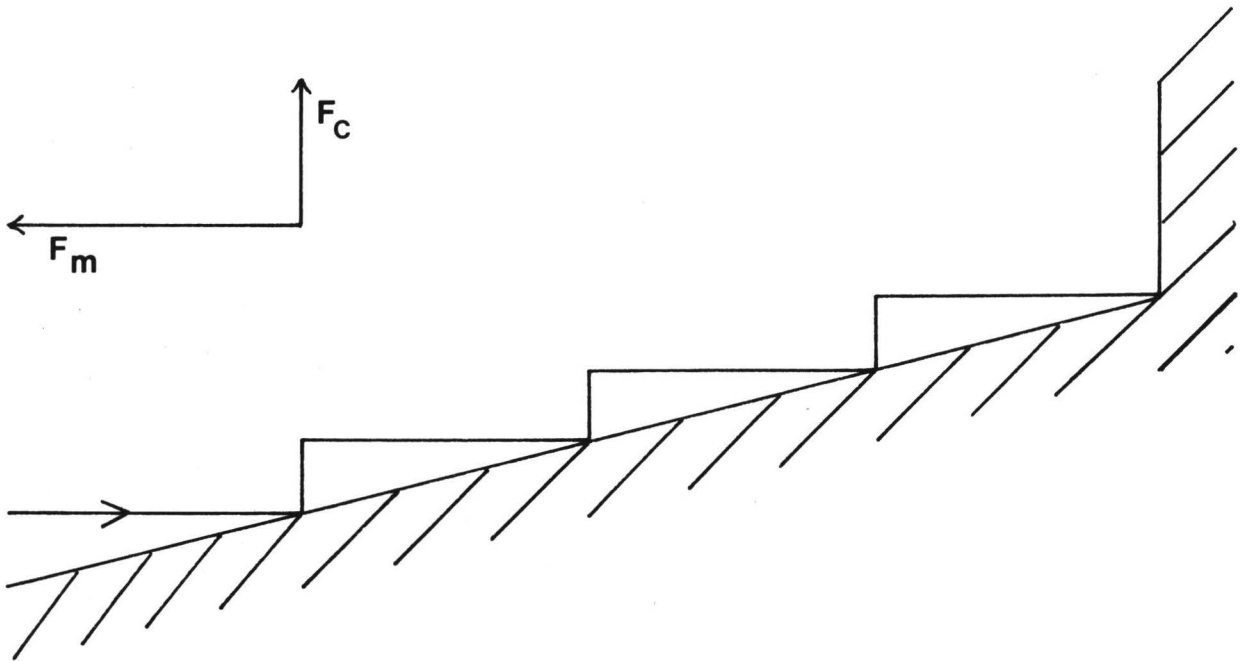


FIGURE 9

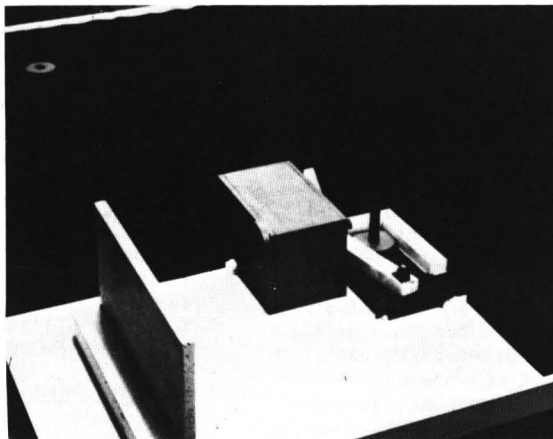


FIGURE 10