Planning techniques find optimal routes

Although robotics and artificial intelligence can be treated as two entirely separate disciplines, there is a good deal of interaction between them. Mark Witkowski looks at the impact of artificial intelligence techniques on robotics.

THERE ARE many possible reasons for applying artificial intelligence techniques to robotics. One is to gain a better understanding of the essential nature of intelligence — why some computations seem clever and worthy of further investigation and others do not, even though they appear more complicated.

Another is to discover new ways of manipulating data which are easier and more natural to write, which increase the efficiency or the applicability of an algorithm to a particular problem. Artificial intelligence has always been something of an assortment of ideas about perception, problem solving, abstraction, generalisation, skilled action, description, language, learning and memory and so on.

The tendency is to investigate those areas in isolation, even though the crudest definition of intelligence would indicate that it is not only the possession of these faculties but their interaction which is of significance.

At the moment, no robot possesses all those faculties but there are a handful which each demonstrate at least one or two to a significant extent.

Fortunately, it is not necessary for a robot to be very intelligent for it to tell us something useful about robot control. The ideas generated in research will slowly find their way to the shop floor and industrial robotics. It is, after all, easier to find a specific solution to a problem once a method of finding solutions in that area is understood.

Construction

Edinburgh University's Freddy system was programmed to construct small wooden toys from their component parts — Ambler et al. (1975) and Barrow and Crawford (1972). Were it not for the fact that this system could start from a situation in which the parts were tipped in a heap on the workbench before assembly commenced, the problem would have been relatively easy.

Furthermore, the algorithm was sufficiently robust to allow the initial pile to contain parts for more than one model of the same or different types, and totally extraneous parts which had to be identified and discarded.

Freddy was a five-degree-of-freedom manipulator in which the gripper could be lowered and raised on a gantry, rotated and closed. X and Y translation of the objects was achieved by moving the workbench. A small vice was fitted to the bench into which objects could be clamped during assembly.

Sensing was provided in the form of proprioceptive co-ordinate feedback, two television cameras, one looking obliquely at the table, the other directly downwards. The gripper was fitted with tactile and force sensing.

The complete assembly process was not totally autonomous — the operator was

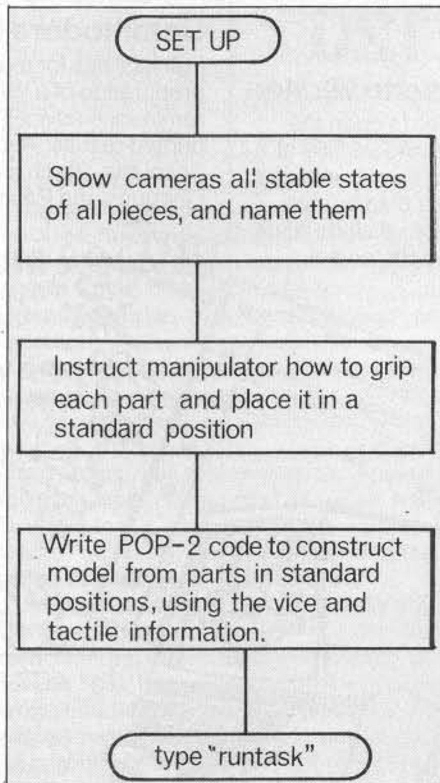


Figure 1. Operator actions.

required to do several things before the robot could be left to assemble models from piles of parts. The automatic part of the program proceeded in two stages.

In the first, parts were isolated from the piles, identified and laid-out in standard locations. This kit of parts would then be assembled using hand-coded routines.

The user had to do three separate programming or teaching operations before the robot was ready to go — figure 1. First, each part of each of the models had to be shown to the system in each of its stable states — the ways it would come to rest if dropped on the table.

That might be repeated several times so that the program could build-up an internal description or representation of the part so that it could be recognised and identified later using only incoming visual sensory data.

Next, the user had to instruct the robot, using a keypad, how to pick-up, rotate and finally deposit in a standard position for assembly each of the parts used in the models. The user had to write some POP-2 code to take the parts from their standard positions and construct the model using the vice to clamp the pieces and tactile sensing to do any close insertion assembly.

POP-2 is the Edinburgh artificial intelligence programming language and not a specific assembly language like WAVE or AL — Burstall, Collins and Popplestone (1971).

Figure 2 shows the automatic part of Freddy's operation — a loop which can be cycled forever. Each time, the most useful operation which can be done in completing the model is executed first. So if everything is complete, the program finishes 1.

Standard

If all the parts required for the model are in their standard positions, the model is assembled, using the pre-defined code, 2. If this was not the case, the cameras are used to explore the table-top. A potential item is a bright region on the dark background — 3.

Once a bright region is located, it must be visually analysed. It will either be a useful item, a piece of the model still needed for the process to continue, in which case it is moved to its standard position — 4.

It could be part of the model but one which duplicates a part already in its standard position, and it must be put to one side — 5.

If there are no regions that can be identified as useful items, the robot sets about the smallest region as a heap — 6. The tactic used is to divide the heap into its individual pieces so they may be identified. The first strategy is to locate visually a protusion from the side of the heap and attempt to pick it up and place it in a clear area for identification.

If for some reason this fails for all the visible protusions, a second tactic is then employed to separate the heap.

The gripper is lowered on to the heap until it touches, thereby defining its height. Then an attempt is made to grab at

the heap, first halfway up and then, if that fails to isolate a single item, at the base.

In the case of a particularly entangled heap, a final attempt is made by ploughing the hand through its centre just above table level. That procedure is not entirely desirable as it causes significant disruption of the work-table lay-out. If the heap is still unrecognisable, it might as well be disposed of — 8.

That portion of Freddy's algorithm is characterised by a number of very useful ideas. First of all, extensive use is made of both visual and tactile feedback and there are many error recovery modes. Everything is checked periodically to make sure it has not moved and that the computer's internal description of the world matches the sensory data — 7.

Most of all, it is very persistent due to the structure of the main control loop — figure 2 — and will work away at objects and heaps until they succumb.

There are also checks to ensure that the proposed action is still applicable. For instance, just before smashing a heap, it checks that the heap is not really a recognisable object which slipped-through. The assembly routines are not as robust. It is the user's responsibility to include such checks as he or she feels appropriate, and if those tests are not made, the assembly may fail in an unexpected way.

Obviously the tactic actions of the layout algorithms are related closely to the types of item they manipulate. The vision routines depend on the objects being lighter in colour than the background, and the objects must be grippable by the hand.

Sensors monitor continually for the unexpected and error recovery was included at many levels. However, there was very little planning involved, actions being made in response to some immediate need. Problem solving and planning is an area where artificial intelligence can really help robotics.

Maze running

Of particular interest to anyone who may be entering the micromouse maze-running competition is the question posed by the exploration and learning of a maze.

The classic method of traversing a maze from some entrance to an exit is to keep touching either the left- or the right-hand wall until the exit is found. That would work for the maze shown in figure 3. You may note, however, that following the right-hand wall leads to the exit a good deal sooner than following the left.

It is, of course, entirely arbitary as to which handedness is to be more efficient. Without knowing something further about the maze, there is no way of telling.

There is, in fact, a particularly nasty catch to the follow-the-wall algorithm — it works only if both start and finish are on an infinite face, that they are joined by a continuous wall.

There is no problem with figure 3 as they are both on the outside wall. Unfortunately, the micromouse competition rules clearly state that the finish will be in the centre of the maze, and so there can be no assumption that the algorithm will terminate.

It is also pointless to take turnings at random, since this would give very slow progress through the maze. It would be worse to change walls at arbitary times. A systematic search of the maze is required. This will not help much for a single timed run but will be very valuable if the maze runner has a second chance.

Tarry's algorithm is useful — Berge (1962). It states that one should never go in the same direction twice along any one edge, nor take the edge from a junction by which one arrived unless no other choice is available. Figure 4 shows the maze in figure 3, depicted in the form of a graph.

Each square in the maze at which a decision can be made is represented by one of the lettered notes, A to N, dead-ends are shown by 'X'. Arcs joining the nodes show the distance between junctions.

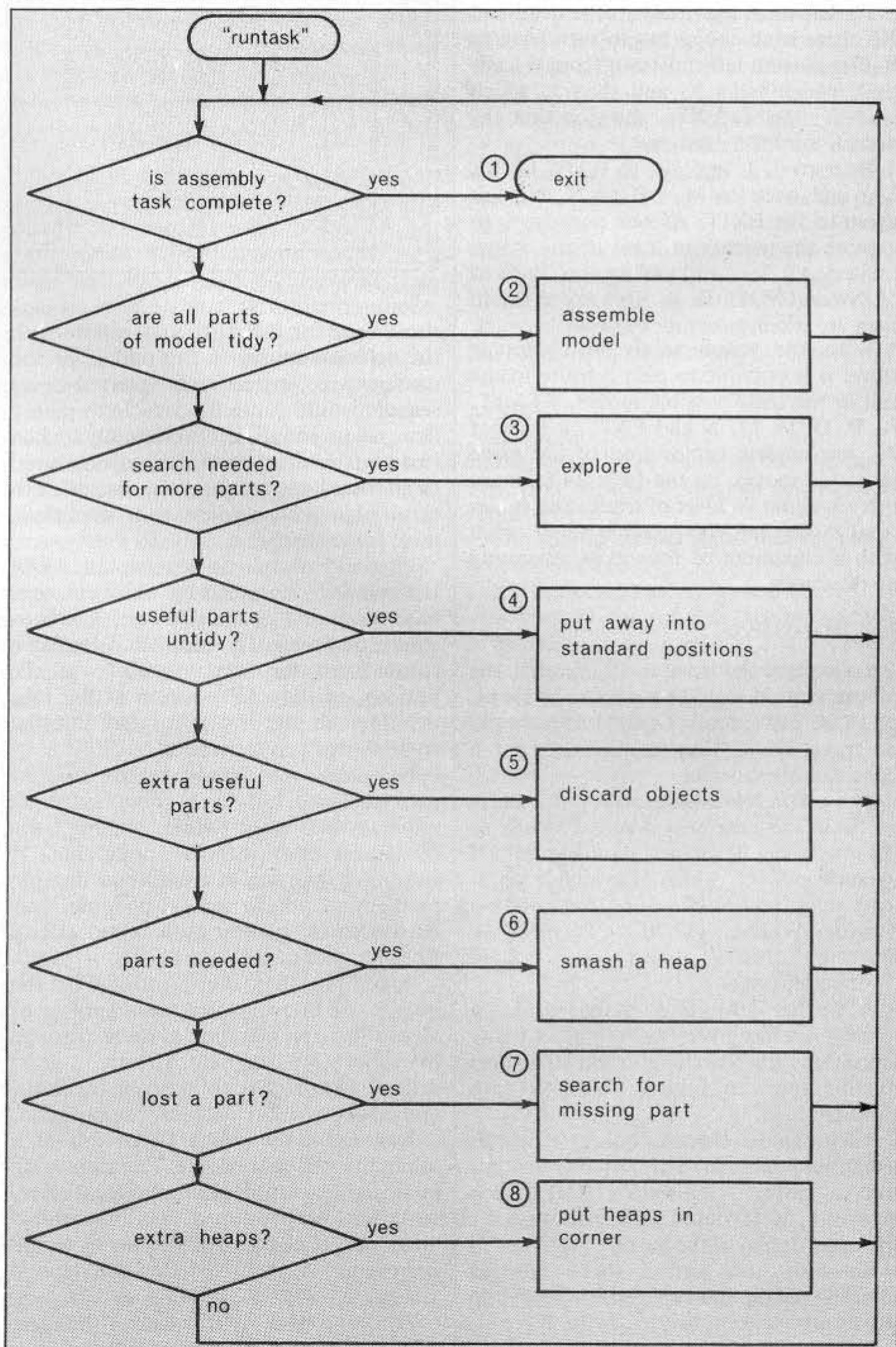
Clearly, with a graph like this, one could explore the maze and choose an optimum route without moving at all. By looking at either the ground plan or the graph, any particular route can be investigated. Following the right-hand wall leads to the exit via:

START(1), A(1), B(1), Xb(1) dead-end so back to B(5), D(3), H(3), M(2), N(2), back to N(5) and then EXIT, for a total of 26 moves.

Following the left-hand wall is altogether worse:

(continued on next page)

Figure 2. Lay-out algorithm.



(continued from previous page)

START(1), A(3), C(4), F(3), Xf(3), F(1), I(1), J(1), K(2), Xkl(2), K(3), Xkr(3), K(1), J(3), L(3), Xl(3), L(3), E(2), G(3), H(3), M(2), Xm(2), M(2), N(5), EXIT, for a total of 64 moves with six dead-ends visited.

The best strategy is to travel through each tunnel and visit each junction in turn, but re-tracing one's steps as little as possible and remembering the internode distance. That must be methodical and some variant of Tarry's algorithm could well be used.

The mouse must first have some way of remembering each of the junctions, probably as an X-Y co-ordinate and then start exploring the maze. As an example, one might turn left unless that tunnel had already been mapped. So from the start there is no choice but to visit A, and the left-most exit goes to C, and thence to F.

F's left-most exit leads to the dead-end Xf, there is no choice but to turn back to F. The current left-most exit from F leads to I, which visits Xi and then J, which visits K, Xkl and Xkr, showing that the node K is itself a dead-end.

Back to J, L and Xl, to E, G, H, M, Xm and back to M, left to N and left again to the EXIT. As our purpose is to explore the maze, not leave it, the exit is treated as a dead-end and we turn back to N, Xn and M. H, D, B, Xb back to B and then A, which takes us back to the start.

With the graph safely in computer store, it is possible to plan a route to the exit in the least possible moves, START, A, B, D, H, M, N and EXIT, a total of 20. A complete exploration of the maze takes 132 moves, on the 14 × 14 ft. maze there is about 700 feet of track, and somewhat more than 600 possible nodes, each with a maximum of four exits, assuming no diagonals.

Exploration

To explore the maze in 10 minutes, the mouse's speed would have to be in excess of 14 in. per second. Open spaces should be traversed as they could represent a considerable shortcut.

The graph representation is particularly useful in this case as it is suited ideally to list processing languages, — Foster (1967) — such as Lisp, which is available on at least three microprocessors, the 6800 — Van der Wateren (1978) — the 6502 — Gardner (1979) and the Z-80 (Softwarehouse).

A further advantage is that artificial intelligence has given rise to a great many algorithms for searching graph structures of this form to find an optimal path through them.

They can be elegant, quick, efficient, exhaustive or heuristicly-driven, according to taste — Nilsson (1971). Each algorithm is favoured in subtle ways by the exact design of the maze.

So with luck and a turbo-charged mouse — in the final analysis there is little substitute for well-directed brute force — a winner will actually reach the exit. Also see Allen and Allen (1979) and Stanfield (1979).

Maze running is a special case of a more general navigational problem that is solved by planning techniques. A mobile robot must operate in the passages and spaces between obstacles without hitting them. Even if the vehicle has an accurate picture of its own position, either by dead-reckoning or some navigational aid, and that of the obstacles it has to avoid, it must still plan a route from its current position to its destination.

In a warehouse, algorithms akin to those used for the maze may suffice with the vehicle running in the middle of the passageway. Any obstacle detected by its sensors would cause the vehicle to plan a new route round it. Presumably, when two such vehicles meet, being too stupid to go round one another, they would both turn, plan a new route and, doubtless, meet somewhere else.

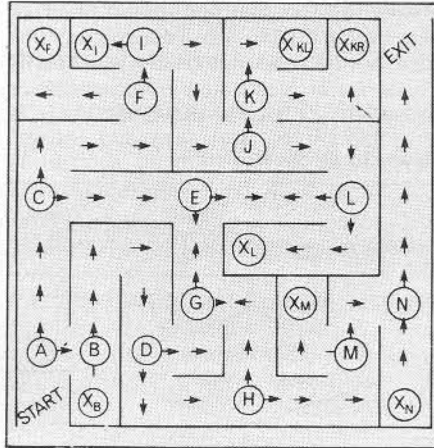


Figure 3. A maze.

Figure 5 shows an open-plan robot environment, bounded by walls but containing a few — five in this case, A to E — square obstacles. The problem is to plan a route from the start position, at the bottom, to the goal position at the top, avoiding all the obstacles, but obeying some shortest path criterion.

Normally, that would be the shortest total distance but in a robot suffering navigational error while turning, the straightest path may be preferable. If computer time was at a premium the first path found, of the several possible, may be chosen or the best path found after a fixed number of seconds.

Assuming that the positions of the objects are known, there are a number of algorithms for planning a route through the robot's environment. Clearly, a good deal of geometry is going to be involved, and hence a good deal of computation.

Any technique which keeps this at a minimum will be welcome. The map could be stored as a topological, graphical representation, perhaps in a two-dimensional array. Each element in the array would correspond directly to a co-ordinate in real space.

For large areas, particularly if there are only a few objects, that will be very cumbersome. Saving only the corner points of the objects would be far more efficient. In planning a minimal route it is desirable to pass by the objects as closely as possible to avoid travelling excess distance.

Computation can be further reduced by treating the robot as a point and by expanding each of the objects it must avoid by an amount equivalent to the radius of a circle which just surrounds the robot.

The result of this expansion is shown in figure 5. Clearly, if a point can navigate round those obstacles, the robot can move around the originals.

The next stage is to build a graph of all the points visible from the current position, and then all the points visible from those new places, and so on. A corner is visible from the current position if a line can be drawn to it without crossing any line which represents the face of an object, i.e., 1.3-1.4.

That could be rather time-consuming even though the routine to test if one line crosses another is minimal. Time could be saved by noting that a good deal of the robot world is invisible from any point as it is occluded by other obstacles. Figure 6 shows such a graph.

There is no need to join nodes at the same depth, 1.n or 2.n and so on, since it is pointless going somewhere in two stages when it is possible to arrive there by a straight line. Each of the arcs shows the length of the line between the two points in question. The underlined number beside each node is the distance which has been travelled to reach it.

Deeper nodes

Where two routes pass through the same point, only the shorter is used to compute distances to the deeper nodes. Eventually, the goal point is reached, or there are no more nodes to expand as the goal was unobtainable anyway.

The distance and route to be taken is now obtained easily from the graph. Searching the graph can proceed in a number of ways. First a breadth search, in which all the first-level nodes are expanded, 1.n, followed by all the second-level nodes, 2.n, then successively deeper nodes.

Searching in this way, the goal node to be found first is 1.6 — 2.4 — Goal, 179. The search would have to proceed to the fifth level to obtain the best route. When there are a large number of nodes, richly interconnected, the search space can become massive in a combination of explosion. However, the combinatorial explosion does not sound the death knell of artificial intelligence problem solvers.

The perfect search strategy is to know some heuristic measure which indicates the most advantageous arc of the many possible. Heuristics are often referred to as rules-of-thumb, extra knowledge or understanding about the problem domain.

A perfect heuristic would lead to a total depth first search, in which one particular successor to a node, rather than its neighbours, is expanded. That would lead directly to the goal.

In reality, a heuristic measure only indicates which of the nodes it might be best to explore. If the search leads to a terminal node, dead-end or one known not to be useful, the search must back-up to a previous node and follow another promising series of arcs.

Possible heuristic measures for searching figure 6 might include expanding the node which has the shortest route back to the start point, or expanding arcs that represent directions that most directly point to the goal position.

Using the co-ordinates of the points, the optimal path START — 1.3 — 2.2 — 3.2 — 4.1 — GOAL can be converted into a LOGO program, which could drive a turtle:

```
TO GOTOGOAL
10 RIGHT 39          (turn 39 degrees right)
20 FORWARD 41        (go 41 units forward)
30 LEFT 41
40 FORWARD 36
50 LEFT 70
60 FORWARD 27
70 RIGHT 24
80 FORWARD 36
90 LEFT 15
100 FORWARD 22
110 END
```

The more general case where the objects to be circumnavigated are not squares but arbitrarily-shaped is nothing like as straightforward. This simple edge expansion is not optimal. In fact, the robot could have squeezed between blocks A and B of figure 5 and if the block had been rounded at the corners to the robot's radius, the solution path would have been totally different. Further details of these algorithms may be found in Lozano-Pérez and Wesley (1979).

Planning and problem solving can be used in generating higher-level, more descriptive plans than those purely for navigation or maze-running. The Shakey robot project at the Stanford Research Institute (SRI) used a problem solver (STRIPS — Stanford Research Institute Problem Solver) to tackle chain of action tasks — Fikes and Nilsson (1971).

Figure 7 shows a typical Shakey environment. A suite of rooms connected by doors to an adjoining corridor contains the robot and a selection of boxes.

The robot can make actions within this world by applying any one of a number of different operators, such as 'goto', 'pushto' or 'gothrudoor'. Whenever there is more than one possible operator, several difficulties arise during planning which were not noticeable with the maze and navigation examples.

Before, only the robot or micromouse changed position. There were no other effects and it was assumed that whenever the robot moved it is no longer where it was and has arrived at its destination.

Environment

During STRIPS planning, even though nothing in the real environment is moved, when it plans to move an object or the robot, the old information in the database about that thing must be removed and replaced with updated information about its new status.

So each time a new node is added to the problem graph by planning to apply an operator, a new version of all the axioms must be generated. That is the essence of the frame-problem: every time you plan an action, the next stage in your plan must assume the world has been changed as a consequence of previous actions.

STRIPS deals with that by having a delete and add list for each of the operators which can be used. The delete list specifies which of the current world model axioms will no longer be true of the world if that operator were to be applied; the add list specifies the axioms which would have to be added after it was used.

A further complication is that operators may only be used if certain conditions are true of the world. The robot may not, for instance, push a box unless it is already next to it. Thus the operator:

gotol(m)

in which the robot moves to place 'm' has the pre-condition:

(4x) [INROOM(ROBOT,x) ∧ LOCINROOM(m,x)]

which states that the robot and the proposed new place for it must both be in the same room. The delete list:

ATROBOT($), NEXTTO(ROBOT,$)

tells the system that wherever '$' the robot was, and whatever it was next to, it will no longer be there after the operator goto(m) is used. The add list:

ATROBOT(m)

is the new information the model requires; the robot will be at 'm'. The operator goto2(m) moves the robot next to the item 'm', which could be, for example, a box or doorpost. Gothrudoor(k,l,m) causes the robot to go through door 'k' from room 'l' into room 'm' and it has the preconditions:

NEXTTO(ROBOT,k)∧CONNECTS(k,l,m)∧ INROOM(ROBOT,l)

The robot must be beside the door 'k'; 'k' must connect room 'l' to room 'm' and the robot must be in room 'l'. The delete list is:

ATROBOT($), NEXTTO(ROBOT,$), INROOM(ROBOT,$)

stating that the robot is neither where it was, next to what it was nor in the same room as before.

The add list simply states that the robot is in the new room:

INROOM(ROBOT,m)

A goal for the robot to achieve, a task or problem to be solved is also couched

(continued on next page)

Figure 4. Graphic representation of maze in figure 3.

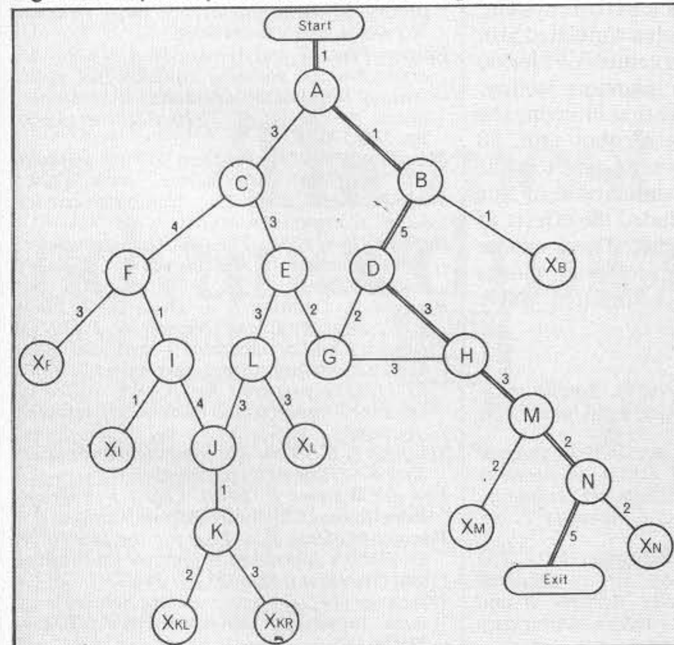
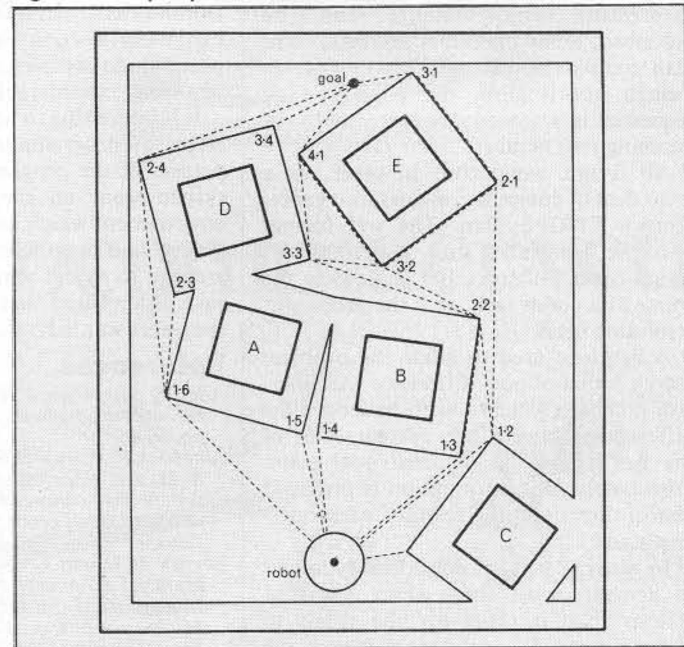


Figure 5. An open-plan robot environment.



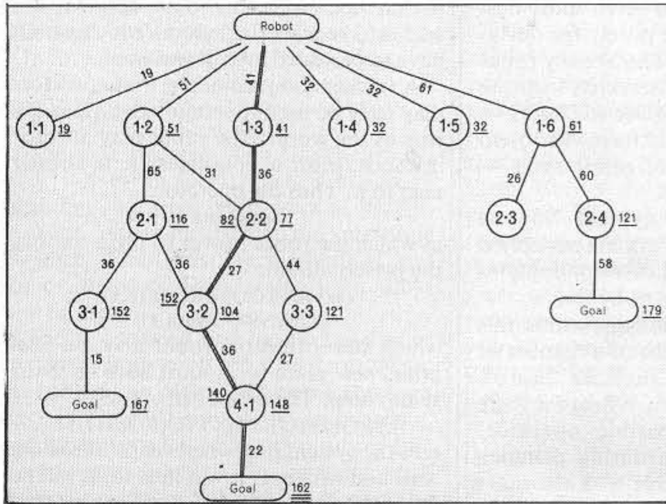


Figure 6. Graph of navigation problem posed in figure 5.

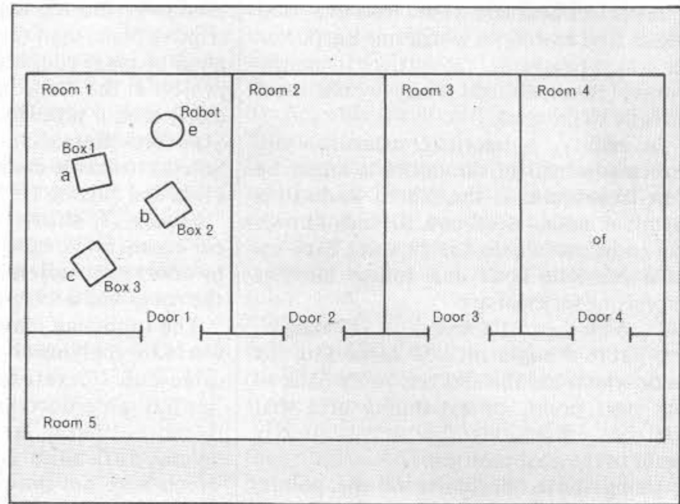


Figure 7. A STRIPS/Shakey world.

(continued from previous page)

in terms of a logic well-formed formula (wff):

NEXTTO(BOX1,BOX2)ΛNEXTTO(BOX2, BOX3)

place box 1 next to box 2 and box 2 next to box 3. Group all three boxes together. The problem solver proceeds by trying to show that the goal wff follows logically from the axioms describing the world and actions by the process of resolution. Strictly speaking, it does exactly the opposite of that — Nilsson (1971) and Kowalski (1979).

Almost as a by-product of that proof the operator list is generated:

goto2(BOX2), pusto(BOX2,BOX1), goto2(BOX2), pusto(BOX3,BOX2)

or the goal wff:

ATROBOT(f) gives:
goto2(DOOR1), gothrudoor (DOOR1, ROOM1, ROOM5),
goto2(DOOR4), gothrudoor (DOOR4, ROOM4,ROOM4),
goto1(f)

The system is clearly far more powerful than either of the previous 'planners'. Interesting environments can be described, many operators can be used to plan complex sequences of actions. Even though not English, the goals can be requested in a reasonably clear, and very unambiguous manner.

All is not wonderful, however, as a great deal of computation goes into generating a STRIPS plan. The wff format must be translated into its equivalent clause form, Nilsson (1971), updating the frame as a major task, as is the process of resolution itself.

A heuristic used to guide the problem search is that of goal difference. An operator is chosen which is likely to reduce the differences between the current state of the world and the required goal state. Fortunately, this information is provided almost directly in the form of each operator's add list.

In general, it takes considerably longer to generate even those short plans of actions than it takes for the robot to execute them. To overcome that to a certain extent, the designers added a facility to store portions of plans made to solve problems, so that they could be recalled and used *en bloc* — Fikes, Hart and Nilsson (1972a) — and also to generalise their stored plans so that they would be applicable as widely as possible.

Furthermore, they looked at the problems introduced by a second active unit in the environment, a second robot, which would change the world without updating the database axioms of the other — Fikes, Hart and Nilsson (1972b).

The lower levels of the Shakey system used a form of route planning similar to the one described earlier. Hardware checks, co-ordinate verification and error recovery, along with many other aspects are all integral in a project of this nature. Some idea of the scope of the Shakey project might be gained from Raphael (1976) or Raphael et al. (1971).

A number of other robot planning systems have been devised which do not involve robots, but simulate their actions on computer terminals. Among them are Doran's pleasure-seeking automaton, Doran (1968), Fahlman's BUILD system, Fahlman (1974), in which a simulated arm would build complex structures of blocks, requiring considerable planning ability.

It is interesting to note that in saving the effort of programming a robot arm, 80 percent of the programming effort in the system went on the simulation of the environment which included the effects of gravity and over-balancing. There was no attempt to model arm trajectories; blocks just disappeared and re-appeared where they were wanted.

References

Allan S and Allan S A (1979). Simple maze traversal algorithms. Byte 4-6, June 1979, pp. 36-46.

Ambler A P, Barrow H G, Brown C M, Burstall R M and Popplestone (1975). A versatile system for computer-controlled assembly. Artificial Intelligence 6-2, Summer 1975, pp. 129-156.

Barrow H G and Crawford G F (1972). The Mark 1.5 Edinburgh robot facility. Machine Intelligence 7 pp.465-480. Meltzer B and Michie D (eds.). Edinburgh University Press. ISBN 0-85224-234-4.

Berge G (1962). The theory of graphs. Great Britain: Methuen & Co.

Burstall R M, Collins J S and Popplestone R J (1971). Programming in POP-2. The Edinburgh University Press. ISBN 0-85224-197-6.

Doran J E (1968) Experiments with the pleasure-seeking automaton. Machine Intelligence 3 pp. 195-216. Michie D (ed.). The Edinburgh University Press. Congress 67-13648.

Fahlman S E (1974) A planning system for robot construction tasks. Artificial Intelligence 5-1, Spring 1974, pp. 1-49.

Fikes R E, Hart P E and Nilsson N J, (1972a), Learning and executing generalised robot plans. Artificial Intelligence 3-4, Winter 1972, pp. 251-288.

Fikes R E, Hart P E and Nilsson N J (1972b). Some new directions in robot problem solving. Machine Intelligence 7 pp. 405-430. Meltzer B. and Michie D. (eds.). Edinburgh University Press. ISBN 0-85224-234-4.

Fikes R E and Nilsson N J (1971). STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2-3/4, Winter 1971, pp. 189-208.

Foster J M (1967). List processing. London/ New York: Macdonald/Elsevier Computer Monographs. SBN 356-02225-0.

Gardner M (1979). The thinking computers language. Practical Computing 2-10, October 1979, pp. 82-84.

Kowalski R (1979). Logic for problem solving. New York: North Holland, Computer Science library, artificial intelligence series (Nilsson N J (ed.)). ISBN 0-444-00365-7.

Lozano-Pérez T and Wesley M A (1979). An algorithm for planning collision-free paths among polyhedral obstacles. Communications of the ACM 22-10 (October 1979) pp. 560-570.

Nilsson N J (1971). Problem solving methods for artificial intelligence. New York: McGraw-Hill Book Co., Computer science series. Congress: 74-136181.

Raphael B (1976). The thinking computer. San Francisco: W H Freeman & Co. ISBN 0-7167-0733-3.

Raphael B, Chaitian L J, Duda R O, Fikes R E, Hart P E and Nilsson N J (1971). Research and applications — artificial intelligence. Semi-annual progress report 7/10/70 to 31/3/71 prepared for NASA, office of advanced research and technology research division.

Stanfield D E (1979). My computer runs mazes. Byte 4-6 (June 1979) pp. 86-99.

Van der Wateren F (1978). Lisp 1.5 programmers' manual. Software documentation.

Van der Wateren F — Lisp for the M6800 in: Dr Dobb's Journal of Computer Calisthenics and Orthodontia No. 28 pp. 24-25.

Winograd (1972). Understanding natural language. Edinburgh University Press ISBM 0-85224-227-1.