LOST IN MAZES

In previous instalments of Workshop we developed the hardware and software to drive a two-motored vehicle and control its direction (see pages 585 and 612). Now we develop an 'intelligent' program that will steer our two-motor vehicle through a maze by selecting the shortest route.

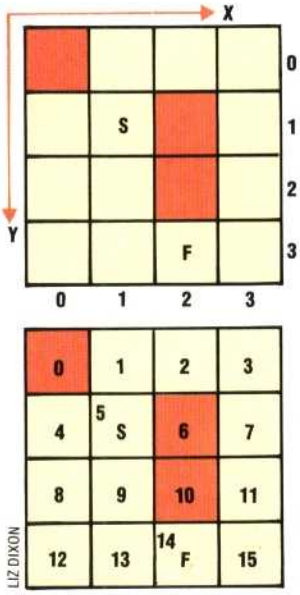
The first stage in constructing a maze is to decide on the area that will constitute the maze. This could be a table top or an area on the floor. The area designated should then be divided into a number of squares, the size of each square being dependent on the size of the vehicle that will be used to negotiate the maze. Each square should be large enough to allow the vehicle to pivot through 360° within a single square. The area can then be marked out as a grid. Objects such as books, cups, or short lengths of wood can then be placed in the area to form the maze.

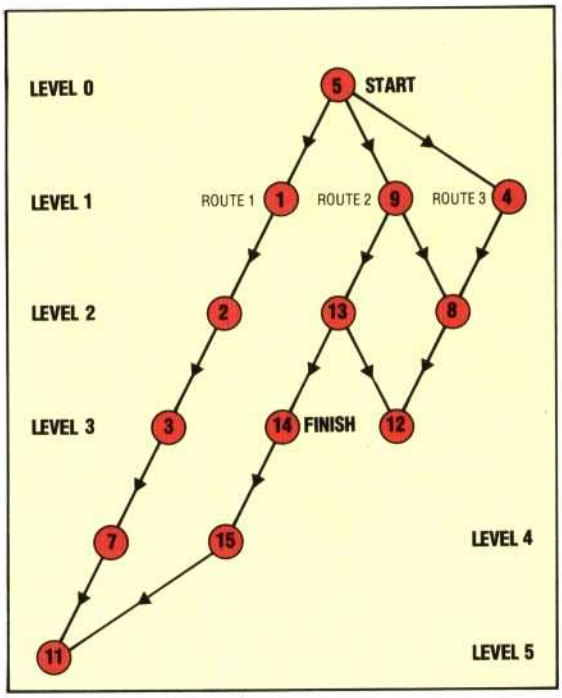
The program requires you to specify the dimensions of the maze, and the locations of squares in the maze that are occupied and those that are free. The easiest method of doing this is to use a binary code: 1 indicating that a square is partially or fully occupied by an object, 0 indicating that the square is free. In order that the maze data does not have to be entered each time the program is run, this information must be written as a series of DATA statements. The final four items of data are the start and finish point co-ordinates. We can imagine the origin of the co-ordinate system as having its origin at the top left corner, the top row being row 0 and the leftmost column being column 0. This maze corresponds to the following DATA statements:

```
0 DATA 4,4:REM DIMENSIONS OF MAZE
  DATA 1,0,0,0,0,0,1,0
1 DATA 0,0,1,0,0,0,0,0
  DATA 1,1:REM COORDS OF START
2 DATA 2,3:REM COORDS OF FINISH
```

Finding a route through a maze does not present many difficulties. We can design a program that will trace a route from the start point, backing out of blind alleys and retracing steps until the finish point is eventually stumbled upon. The eventual route found (without the detours into blind alleys) may or may not be the shortest possible route. If we want to find the *best* route between the start and finish points then we must adopt a method that tests all possible routes between the two points. It is worth noting that our program interprets 'best route' as the route that uses the least number of squares.

Two-Way Drive
The maze solving program interprets the maze in two ways. As the maze is read in from data statements it is stored in a two-dimensional array, the start and finish points also being held initially as co-ordinates. In order to solve the maze the program must treat each square in the maze as a 'node' in a tree. Rather than using the initial co-ordinate system, each square is therefore numbered in order, starting at the top left-hand corner of the maze





Tree Structure
Before the best route through the maze can be found, a 'tree' representing the relationships between squares within the maze has to be constructed. Each node is considered in turn, creating lower levels of nodes. Level 1 nodes are one square removed from the start; level 2 nodes are two squares removed from the start, and so on. It is fairly straightforward for us to draw the tree, but implementing this structure in BASIC is more difficult

We can make the task of testing each route easier by enforcing a structure on the maze data that represents relationships between squares. The data structure that most lends itself to this application is a hierarchical tree. Beginning with the start point as the 'root' of the tree, we can build up a second generation of squares (or 'nodes') that are one square removed from the root. A third generation of nodes can be built from these second generation nodes, and so on. We can draw a tree for any maze by numbering each square and following the rule that descendants from any node are drawn from left to right in the order North, East, South and West of the parent node in the original maze.

This simple maze can be solved in five ways, without retracing one's steps. Three possible solutions are shown above, as routes through the tree and as actual routes through the maze. It is obvious to us that route 2 is the shortest route, but this is because we are able to evaluate the tree laterally; that is we can consider the maze as a whole. The computer must solve the tree in a linear way, taking each possible route systematically until the finish node is found or a blind alley is reached. In the first case a record of the sucessful route must be kept; in the second the path taken must be marked as a blind alley before restarting from the root node. The program will continue to travel through the tree until all branches from the root node have been tried.

BASIC does not lend itself readily to search algorithms of this type and the programming can