```
TRY(i1,j1,i2,j2)   is there a path from i1,j1 using i2,j2?
   IF (maze(i1,j1) is a wall)
      return(NO)
   IF (maze(i2,j2) is a wall)
      return(NO)
   IF (we've been to i2,j2 before)
      return(NO)
   remember that we've been to i1,j1
   put i1,j1 on the path tentatively
   IF (i2,j2 is at edge of maze)
      put i2,j2 on path too
      return(YES)
   ELSE IF (path from up cell adjacent to i2,j2)
      return(YES)
   ELSE IF (path from right cell adjacent to i2,j2)
      return(YES)
   ELSE IF (path from down cell adjacent to i2,j2)
      return(YES)
   ELSE IF (path from left cell adjacent to i2,j2)
      return(YES)
   ELSE
      take i1,j1 off the path
      return(NO)
```

Each test of the form

```
ELSE IF (path from ... cell adjacent to i2,j2)
```

is performed by calling TRY. In this way, TRY handles only a small part of the problem directly, then calls itself recursively when necessary to handle the rest.

Finally, we must consider the question of data representation. We have to record things like whether we've been to a cell before, and also what the actual path is. One simple possibility is to record in an array STATE whether we've been to a cell before or not. The array STATE contains two states, "used" (meaning we've looked at this square before), and "free" (meaning we haven't). We could also use STATE to record the path as it is found, or (as we choose here), use two linear arrays IPATH and JPATH to record the coordinates of the path as it is found. This latter organization takes a bit more code, but is substantially faster, since we never follow a blind alley more than once.

Putting all of these pieces together makes a large program, but no larger than the original, and markedly easier to follow.