Mazes

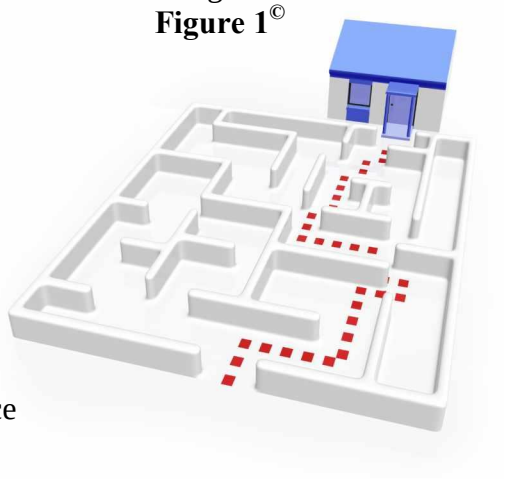
By Dr K R Bond

From course material supplied by Educational Computing Services Ltd

Travelling by car to unfamiliar places is greatly aided by GPS-based route planners that can be purchased for less than £100 such as TomTom[Ê] . Route finding goes back centuries to Cretan times when Theseus destroyed the Minotaur residing in the twisting maze of the labyrinth at Knossos.

Route finding offers plenty of scope for exploring several useful concepts of Computer Science such as abstraction, data modelling, algorithms and algorithm efficiency, record structures, arrays, lists, recursion, stacks, objects, procedure calling and graphs. Figure 1 shows a typical route finding task, finding a route through a maze from entrance to exit, e.g. house to exit from the "maze garden".

Figure 1[©]



How could the junctions, entrance, exit and pathways between these be represented if the task was delegated to a computer program?

First let's abstract away unnecessary detail and record the maze in a representation called a graph using the following rules:

Vertices or Nodes

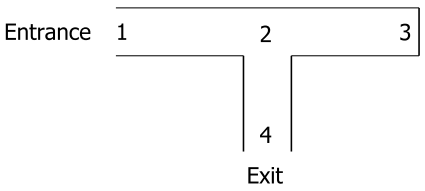
1. Vertex for a starting point, i.e. entrance
2. Vertex for a finishing point, i.e. exit
3. Vertices for all dead ends
4. Vertices for all the points in the maze where more than one path can be taken, i.e. junction

Edges

Connect the vertices according to the paths in the maze.

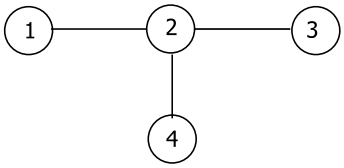
To keep the description as simple as possible consider the "maze" in Figure 2 which has one entrance labelled 1, one dead end labelled 3, one exit labelled 4 and one junction labelled 2.

Figure 2



The graph that models this is shown in Figure 3. The circles represent vertices or nodes and the interconnecting lines, edges.

Figure 3



How can this graph be represented for processing by route finding software?

We can use an adjacency list as shown in Table 1.

Table 1

Vertex	Adjacent vertices
1	2
2	1, 3, 4
3	2
4	2

Vertex 1 is connected to vertex 2. Vertex 2 is connected to vertices 1, 3 and 4, etc.

This table can be represented by a one-dimensional array of records with each record storing the corresponding adjacent vertices, the first record 2, the second record 1, 3, 4 and so on.

[Ê] Registered trademark of TomTom International

[©] Royalty-free stock iStockphoto LP iStock_000003304776Medium.jpg standard licence

The structures for this are shown in Table 2.

Table 2

```
ListType = Array[1..MaxNoOfAdjacentVertices] Of Integer;
VertexType = Record
               List : ListType;
               NoOfListVertices : Integer;
               State : String;
             End;
GraphType = Array[1..MaxNoOfVertices] Of VertexType;
```

Table 3 shows how a Graph array of the type GraphType would be initialized for the graph in Figure 3. The state field is needed so that when route finding the state of a vertex can be changed from "undiscovered" to "discovered".

Table 3

```
Graph[1].List[1] Ŷ 2; Graph[1].NoOfListVertices Ŷ 1; Graph[1].State Ŷ 'undiscovered';
Graph[2].List[1] Ŷ 1; Graph[2].List[2] Ŷ 3; Graph[2].List[3] Ŷ 4; Graph[2].NoOfListVertices Ŷ
3;
Graph[2].State Ŷ 'undiscovered';
Graph[3].List[1] Ŷ 2; Graph[3].NoOfListVertices Ŷ 1; Graph[3].State Ŷ 'undiscovered';
Graph[4].List[1] Ŷ 2; Graph[4].NoOfListVertices Ŷ 1; Graph[4].State Ŷ 'undiscovered';
```

A route from vertex 1 to vertex 4 can be found using the recursively defined algorithm shown in Table 4, coded as procedure **FindRouteToGoal** with signature:

```
FindRouteToGoal(Var Graph : GraphType; CurrentVertex :
Integer;  GoalVertex : Integer; Var Stack : StackType);
```

The **Var** refers to the parameter passing mechanism known as "call by address".

Procedure **FindRouteToGoal** is called initially with **CurrentVertex = 1** and GoalVertex = 4.

Table 4

```
Graph[CurrentVertex].State Ŷ 'discovered';
Stack.Push (CurrentVertex);
If CurrentVertex = GoalVertex
  Then
    While Not Stack.Empty
      Do
        {
          Node Ŷ Stack.Pop;
          Writeln(Vertex);
        }
  Else
    If Graph[CurrentVertex].NoOfListVertices <> 0
      Then
        For i Ŷ 1 To Graph[CurrentVertex].NoOfListVertices
          Do
            If (Graph[Graph[CurrentVertex].List[i]].State
                  = 'undiscovered')
              Then
                {
                  Vertex Ŷ Graph[CurrentNode].List[i];
                  FindRouteToGoal(Graph, Vertex, GoalVertex, Stack);
                  Stack.Pop;
                }
```

Each time this procedure is called the number of the current vertex is pushed onto the stack as shown in Figure 4. If a trial path fails to reach the goal vertex the stack is popped until a path not yet explored can be reached. If the goal vertex is reached, the stack will contain the vertices for a route between the starting vertex and the goal vertex but in the reverse order. This route is printed out before the execution of the procedure halts. If the goal cannot be reached the procedure will halt with the stack empty.

Figure 4

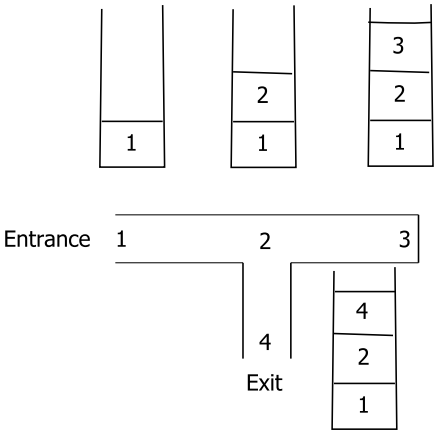
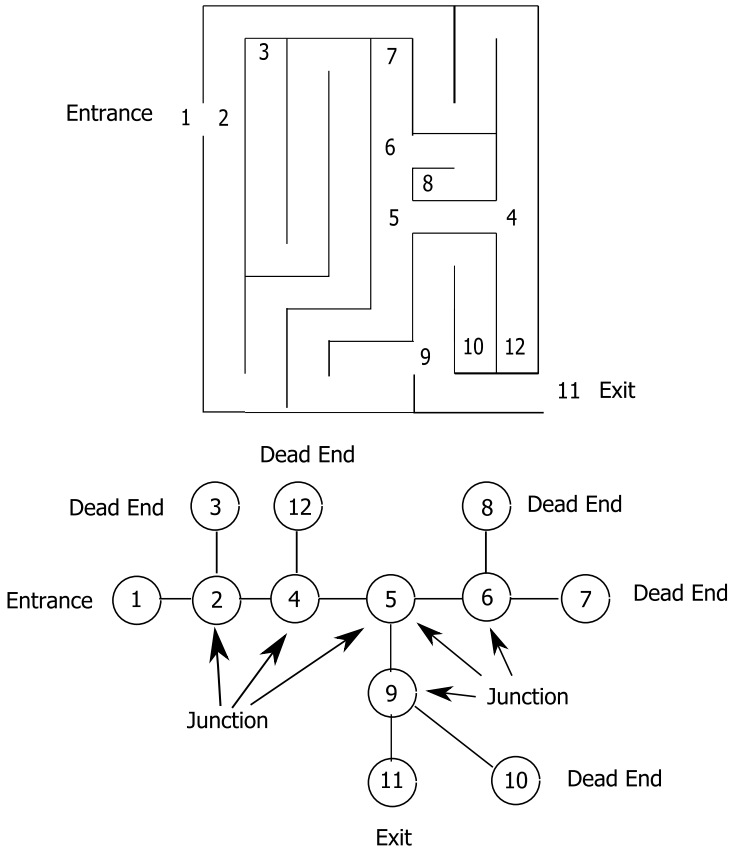


Figure 4 shows the state of the stack as the pathways 1-2, 1-2-3 and 1-2-4 are tried in turn.

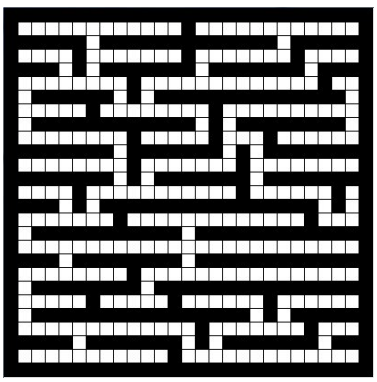
Figure 5 shows a more complex maze and its equivalent graph. Sketching the state of the stack as the algorithm in Table 4 is hand traced is a useful exercise for students to try.

Figure 5



The work here could form the basis of several projects. For example, one such project could generate mazes of specified dimensions as shown in Figure 6 which could then be route searched using the algorithm given in Table 4. One application seen in project work is the design of climbing walls. There are plenty others.

Figure 6



This article is based on the content of an Educational Computing Services Ltd course on teaching AQA A Level Computing and preparing candidates for COMP4 project work.