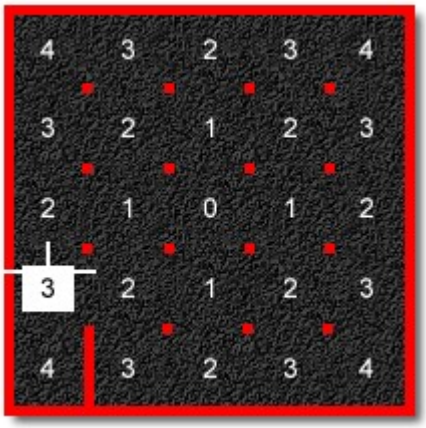
MicroMouseInfo.com
Hints, Ideas, Inspiration for Mice Builders

Software suggestions

The flood-fill algorithm

The flood-fill algorithm involves assigning values to each of the cells in the maze where these values represent the distance from any cell on the maze to the destination cell. The destination cell, therefore, is assigned a value of 0. If the mouse is standing in a ce with a value of 1, it is 1 cell away from the goal. If the mouse is standing in a cell with a value of 3, it is 3 cells away from the goal. Assuming the robot cannot move diagonally, the values for a 5X5 maze without walls would look like this:



Of course for a full sized maze, you would have 16 rows by 16 columns = 256 cell values. Therefore you would need 256 bytes to store the distance values for a complete maze.

When it comes time to make a move, the robot must examine all adjacent cells which are not separated by walls and choose the one with the lowest distance value. In our example above, the mouse would ignore any cell to the West because there is a wall, and it would look at the distance values of the cells to the North, East and South since those are not separated by walls. The cell to the North has a value of 2, the cell to the East has a value of 2 and the cell to the South has a value of 4. The routine sorts the values t determine which cell has the lowest distance value. It turns out that both the North and East cells have a distance value of 2. That means that the mouse can go North or East and traverse the same number of cells on its way to the destination cell. Since turning would take time, the mouse will choose to go forward to the North cell. So the decision process would be something like this

Decide which neighboring cell has the lowest distance value:

Is the cell to the North separated by a wall?
Yes -> Ignore the North cell
No -> Push the North cell onto the stack to be examined

Is the cell to the East separated by a wall?
Yes -> Ignore the East cell
No -> Push the East cell onto the stack to be examined

Is the cell to the South separated by a wall?
Yes -> Ignore the South cell
No -> Push the South cell onto the stack to be examined

Is the cell to the West separated by a wall?
Yes -> Ignore the West cell
No -> Push the West cell onto the stack to be examined

Pull all of the cells from the stack (The stack is now empty)
Sort the cells to determine which has the lowest distance value

Move to the neighboring cell with the lowest distance value.

Now the mouse has a way of getting to center in a maze with no walls. But real mazes have walls and these walls will affect the distance values in the maze so we need to keep track of them. Again, there are 256 cells in a real maze so another 256 bytes will be more than sufficient to keep track of the walls. There are 8 bits in the byte for a cell. The first 4 bits can represent the walls leaving you with another 4 bits for your own use. A typical cell byte can look like this:

Bit No.	7	6	5	4	3	2	1	0
Wall					W	S	E	N

Remember that every interior wall is shared by two cells so when you update the wall value for one cell you can update the wall value for its neighbor as well. The instructions for updating the wall map can look something like this:

***Update the wall map**:*

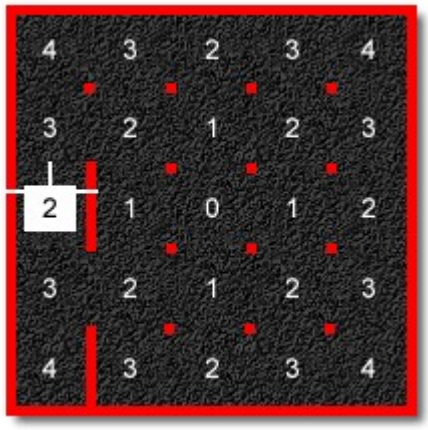
Is the cell to the North separated by a wall?
Yes -> Turn on the "North" bit for the cell we are standing on and
Turn on the "South" bit for the cell to the North
No -> Do nothing

Is the cell to the East separated by a wall?
Yes -> Turn on the "East" bit for the cell we are standing on and
Turn on the "West" bit for the cell to the East
No -> Do nothing

Is the cell to the South separated by a wall?
Yes -> Turn on the "South" bit for the cell we are standing on and
Turn on the "North" bit for the cell to the South
No -> Do nothing

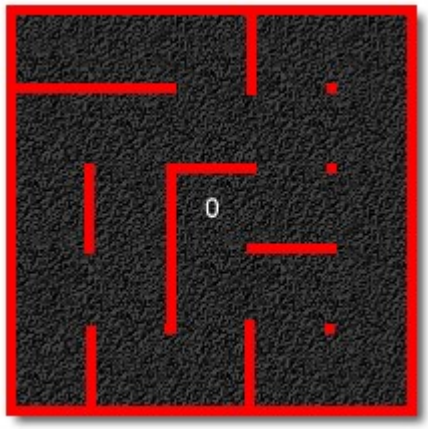
Is the cell to the West separated by a wall?
Yes -> Turn on the "West" bit for the cell we are standing on and
Turn on the "East" bit for the cell to the West
No -> Do nothing

So now we have a way of keeping track of the walls the mouse finds as it moves about the maze. But as new walls are found, the distance values of the cells are affected so we need a way of updating those. Returning to our example, suppose the mouse has found a wall.

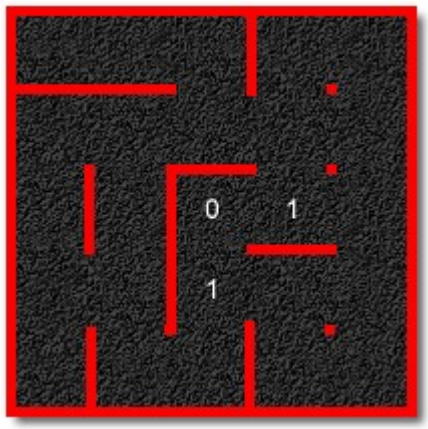


We cannot go West and we cannot go East, we can only travel North or South. But going North or South means going up in distance values which we do not want to do. So we need to update the cell values as a result of finding this new wall. To do this we "flood" the maze with new values.

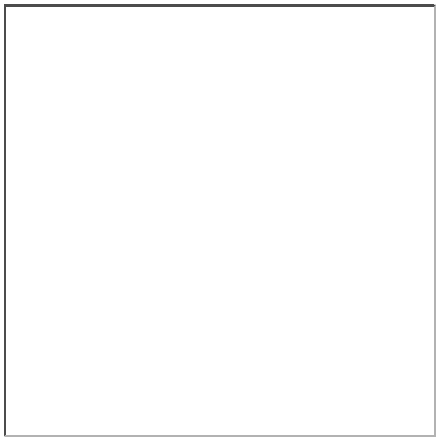
As an example of flooding the maze, let's say that our mouse has wandered around and found a few more walls. The routine would start by initializing the array holding the distance values and assigning a value of 0 to the destination cell:



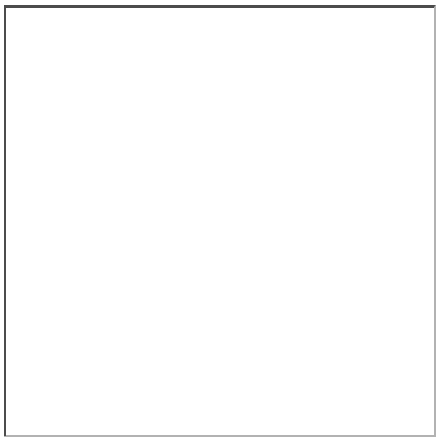
The routine then takes any open neighbors (that is, neighbors which are not seperated by a wall) and assigns the next highest value, 1:



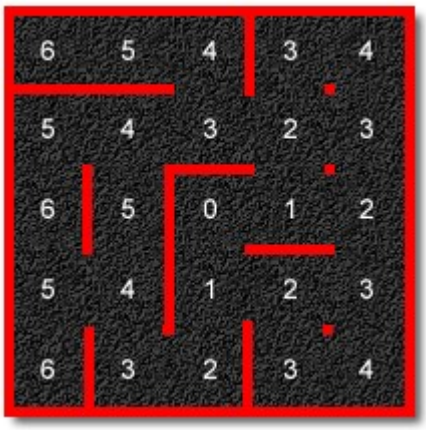
The routine again finds the open neighbors and assigns the next highest value, 2:



A few more iterations:



This is repeated as many times as necessary until all of the cells have a value:



Notice how the values lead the mouse from the start cell to the destination cell through the shortest path.

The instructions for flooding the maze with distance values could be:

Flood the maze with new distance values:

Let variable Level = 0
Initialize the array DistanceValue so that all values = 255
Place the destination cell in an array called CurrentLevel
Initialize a second array called NextLevel

```
Begin:

Repeat the following instructions until CurrentLevel is empty:

        {
        Remove a cell from CurrentLevel
        If DistanceValue(cell) = 255 then

                let DistanceValue(cell) = Level and
                place all open neighbors of cell into
                NextLevel

        End If
        }

The array CurrentLevel is now empty.

Is the array NextLevel empty?
No ->

        {
        Level = Level +1,
        Let CurrentLevel = NextLevel,
        Initialize NextLevel,
        Go back to "Begin:"
        }

Yes -> You're done flooding the maze
```

Summary:

The flood-fill algorithm is a good way of finding the shortest (if not the fastest) path from the start cell to the destination cells. You will need 512 bytes of RAM to implement the routine: one 256 byte array for the distance values and one 256 array to store the map of walls.

Every time the mouse arrives in a cell it will perform the following steps:

(1) Update the wall map

(2) Flood the maze with new distance values

(3) Decide which neighboring cell has the lowest distance value

(4) Move to the neighboring cell with the lowest distance value