IEEE MICROMOUSE

Workshop Week 11:

Flood Fill

Objective:

Understand what the maze coordinates and values are. Use the values of the maze cells to perform path planning using flood fill. Understand what variable pointers in the C programming language are, and why we would want to use them

Background information:

Last week we learned the basics on how to represent a maze and a mouse inside the Arduino's memory. We also learned that we could visualize the maze and mouse by printing the output into a serial terminal. In today's workshop, we will learn how to use the maze and mouse software to run the flood fill algorithm and find the shortest path to the center of the maze.

We will be using the well-known flood fill algorithm to find the center of the maze. Flood fill was initially used in computer graphics to fill bounded sections with a given color. If you ever used the bucket tool in paint, you know what the algorithm does. People discovered that the same algorithm could be used to compute distances from one cell to another, and they used it to quickly recompute cell values whenever they did not agree with the map (wall in between cells).

Later, it was found that the flood fill algorithm did not have to be run on every maze cell, and it only needed to be run when the current cell is one less than the lowest valued open neighbor. This dramatically increased the speed in which the mouse could find a valid path, so this is the algorithm that most micromouse enthusiasts use.

Here is the pseudo-code for the flood fill algorithm:

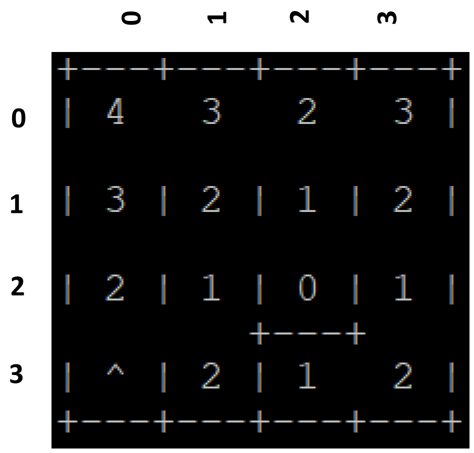
```
void flood_fill(int x, int y)
{
    /*1. Push the current cell location into a stack*/

    /*2. Repeat the following until the stack is not empty*/

        /*a. Pop a cell from the top of the stack*/
        /*b. Find the minimum open neighbor value*/
        /*c. If the minimum open neighbor value is not equal to the current cell value -1*/
            /*i. Replace current cell value with minimum open neighbor value + 1*/
            /*ii. Push all neighbors into the stack*/
}
```

This code might look daunting, but we are going to dissect it one chunk at a time. We are also going to be required to learn how to use a stack and C pointers. Ultimately, we will place our flood fill code inside the maze.cpp and maze.h source files, so that we can access it from our sketch

Maze Coordinates

Last workshop did not discuss the coordinate convention that the maze follows, so let’s address that in this section.

The maze coordinates are as follows:

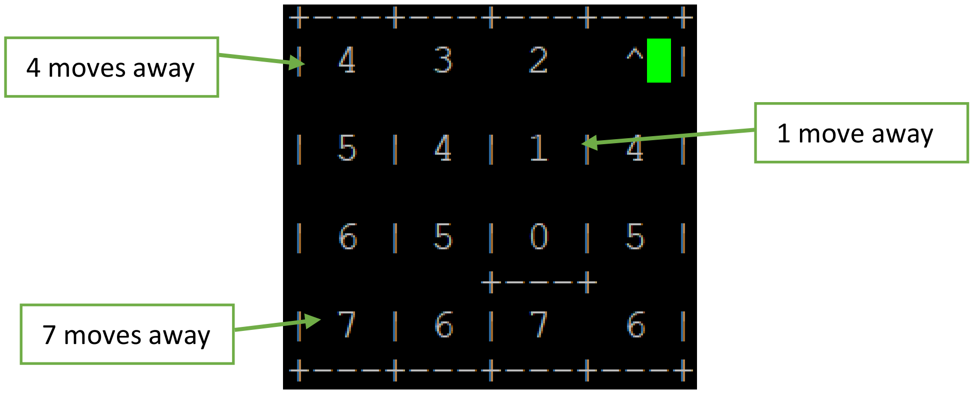


```
+---+---+---+---+
| 4   3   2   3 |

| 3 | 2 | 1 | 2 |

| 2 | 1 | 0 | 1 |
        +---+
| ^ | 2 | 1   2 |
+---+---+---+---+
```

Remember that the mouse position always starts at one of the maze corners, so you must make sure that your software reflects the initial position of the mouse

Maze cell values

The maze cell values tell you the number of moves we have left until we reach the center of the maze, or the target cell with value 0. For example, look at the maze below and see that every cell value tells you exactly how far, or how many moves you still have left in order to reach the 0-value cell



```
+---+---+---+---+
| 4   3   2   ^ |

| 5 | 4 | 1 | 4 |

| 6 | 5 | 0 | 5 |
        +---+
| 7 | 6 | 7   6 |
+---+---+---+---+
```

You can see that by always moving towards the lowest cell value we can reach the center of the maze. In actual code, we will always have the mouse choose the open neighbor with the lowest cell value and move towards it. Pretty neat right?!

C Pointers and C structure pointers

Before tackling the flood fill algorithm, we need to get some new C functionality and syntax out of the way. First, the cells of the maze are represented as C data structures, which have the following syntax:

```c
typedef struct
{
  uint8_t value;
  uint8_t pos;
  uint8_t walls;
}cell_t;
```

A C data structure is nothing more than a collection of variables with an alias, in this case the alias is cell_t. The maze can now be represented as a 2-dimensional array of cells as follows:

```c
cell_t maze[8][8];
```

Now, you might be thinking, why is this useful? Why would we ever want to use this? Well, the alternative is to declare multiple 2-dimensional arrays for the cell values, walls and position such as this:

```c
uint8_t value[8][8]
uint8_t pos[8][8]
uint8_t walls[8][8]
```

Now we must worry about 2 more 2-dimensional arrays, which can lead to confusion and loss of meaning on what each array does. That is why something is useful to think in terms of C structures rather than individual variables.

For the next section, we need to access and modify the values of the cells. To do so we need to access its member variables. Because of the way the stack is designed, we need to understand how to access the structure members for a C pointer.

So, what is a C pointer? Basically, is a variable that stores the address of another variable. Now that sounds abstract, and the application of a C pointer is often not clearly explained, so allow us to explain how pointers are useful in this case.

For the maze, we already have the cells in memory by declaring the previous 8x8 cell array. For accessing a specific cell, we can do something like this:

```c
cell_t current_cell = maze[0][0];
```

This is fine and it will work, but is not memory efficient, since the compiler will create a copy of the cell_t data structure, and that variable consumes 3 bytes of data (1 for value, 1 for walls and 1 for position). Now imagine that we have to push 100 cell_t into a software stack, that will consume 3 bytes * 100 = 300 bytes of memory. That is a lot for our tiny Arduino, which only has 2048 bytes of memory.

The alternative is to use a cell_t pointer variable as follows:

```
cell_t *current_cell = &maze[0][0];
```

Now this pointer only consumes 2 bytes of memory, so it goes without saying that this is an improvement from the previous implementation. Another thing to point out, is that the pointer memory size requirements will stay fixed at 2 bytes regardless of how big the data structure is.

To access the members of a C data structure pointer, we must write this code

```
cell_t *current_cell = &maze[0][0];
uint8_t value = current_cell->value;
```

Not so difficult right?

Software Stack

As you might have noticed, we are going to need a software stack to write the flood fill algorithm. A stack is a data structure algorithm that performs Last In-First Out (LIFO) data access. For the sake of time, a software stack has been implemented for you to use in your flood fill software. You must update the micromouse workshop sketch template to the latest version, or download the stack.cpp and stack.h files and place them inside your sketch.

To push a cell into the stack, use this code:

```
cell_t *current = &maze[0][0];
stack_push(current);
```

To pop a cell from the stack, use this code:

```
cell_t *current;
stack_pop(current);
```

Flood fill

With all that information out of the way, let’s tackle the flood fill algorithm. First let’s write a prototype function inside the maze.cpp file

```
/*FLOOD FILL CODE GOES HERE****************************************************/
void flood_fill(int x, int y)
{

}
/*FLOOD FILL ENDS HERE*********************************************************/
```

Next let’s fill out the function with some psudo-code and actual C code that we can start filling out:

```
/*FlOOD FILL CODE GOES HERE****************************************************/
void flood_fill(int x, int y)
{
  /*Neighbor offsets for all orientations*/
  int8_t neighbor_offset[4][2] =
  {
    {0, -1},
    {1,  0},
    {0,  1},
    {-1, 0},
  };

  /*Pointer to data structure.
  See https://overiq.com/c-programming-101/pointer-to-a-structure-in-c/ */
  cell_t *current;
  cell_t *neighbor;

  /*Get current cell from maze. FILL THIS OUT*/


  /*Push current cell to the stack. FILL THIS OUT*/


  /*Keep running until the stack is not empty. FILL THIS OUT*/
  while()
  {
    /*Pop cell from the stack. FILL THIS OUT*/

    /*If this is the target cell (value 0), then break out of this loop. FILL THIS OUT*/

    /*Get position from popped cell*/
    uint8_t cx = GET_X_POS(current->pos);
    uint8_t cy = GET_Y_POS(current->pos);

    /*Find the open neighbor with the lowest value*/
    uint8_t minimum = 255;
    for(int i = 0; i < 4; i++)
    {
      if((current->walls & (1 << i)) == 0)
      {
        /*Get neighborhs x and y coordinates*/
        int8_t nx = cx + neighbor_offset[i][0];
        int8_t ny = cy + neighbor_offset[i][1];

        /*Get neighbor cell pointer from the maze*/
        neighbor = maze_get_cell(nx, ny);
        /*If the neighbor is less than the minimum,
         * then set this cell value as the new minimum*/
        if(neighbor->value < minimum)
        {
          /*FILL THIS OUT*/
        }
      }
    }

    /*If the current cell value is not one less
     * than the minimum ... FILL THIS OUT
     */
    if()
    {
      /*Increment the current cell value by the minimum + 1. FILL THIS OUT*/

      /*Push all neighbors into the stack*/
      for(int i = 0; i < 4; i++)
      {
        /*Get neighborhs x and y coordinates*/
        int8_t nx = cx + neighbor_offset[i][0];
        int8_t ny = cy + neighbor_offset[i][1];

        /*Check that cell is not out of bounds*/
        if((nx > 0)                 &&
           (nx < (MAZE_WIDTH - 1))  &&
           (ny > 0)                 &&
           (ny < (MAZE_HEIGHT - 1)))
        {
            /*Get neighbor cell pointer from the maze*/


            if(neighbor->value != 0)
            {
              /*Push the neighbor into the stack if the value
               * is not equal to 0. FILL THIS OUT*/

            }
        }
      }
    }
  }
}
 /*FLOOD FILL ENDS HERE*********************************************************/

```

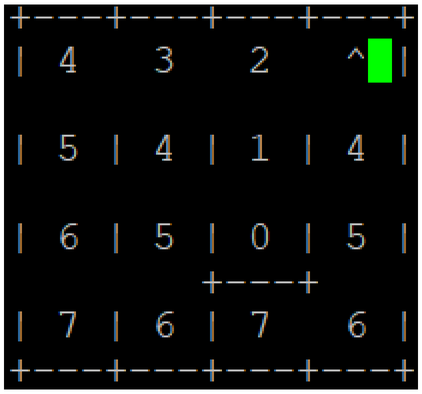
Your job now is to fill out the missing sections of the flood fill program so that it compiles cleanly. Then on your main sketch write this code to test your flood fill algorithm. The code will set up a test maze with walls and intial values. The intial values assume that there are no walls in between the cells, so it is the job of the flood fill algorithm to find the correct values given that are walls in between the cells.

```
#include "micromouse.h"
#include "maze.h"

void setup()
{
  /*Enable serial port for bluetooth*/
  Serial.begin(9600);

  /*Initialize maze*/
  maze_init(MAZE_WIDTH/2, MAZE_HEIGHT/2);
  mouse_set_x(0);
  mouse_set_y(MAZE_HEIGHT - 1);

  /*First column*/
  maze_set_wall(0, 3, EAST);
  maze_set_wall(0, 2, EAST);
  maze_set_wall(0, 1, EAST);

  /*Second column*/
  maze_set_wall(1, 3, EAST);
  maze_set_wall(1, 2, EAST);
  maze_set_wall(1, 1, EAST);

  /*Third column*/
  maze_set_wall(2, 1, EAST);
  maze_set_wall(2, 2, EAST);
  maze_set_wall(2, 2, SOUTH);

  maze_print();
  flood_fill(mouse_get_x(), mouse_get_y());
  maze_print();
}

void loop()
{
}

```

Notice that you can run the flood fill algorithm in any cell on the maze. Play around with this function by passing different x and y values into your flood fill function. See how the maze values change according to the walls

Exercise:

Use the code from workshop 10 to move the mouse around the maze. Explore the maze several times so that the flood fill finds the values of the test maze so that it matches the values from the maze below



```
+---+---+---+---+
| 4   3   2   ^█|

| 5 | 4 | 1 | 4 |

| 6 | 5 | 0 | 5 |
        +---+
| 7 | 6 | 7   6 |
+---+---+---+---+
```

Figure 1 Final solved maze

Pseudo-code to explore and solve the maze

```
#include "micromouse.h"
#include "maze.h"

void setup()
{
  /*Enable serial port for bluetooth*/
  Serial.begin(9600);

  /*Initialize maze*/
  maze_init(MAZE_WIDTH/2, MAZE_HEIGHT/2);
  mouse_set_x(0);
  mouse_set_y(MAZE_HEIGHT - 1);

  /*First column*/
  maze_set_wall(0, 3, EAST);
  maze_set_wall(0, 2, EAST);
  maze_set_wall(0, 1, EAST);

  /*Second column*/
  maze_set_wall(1, 3, EAST);
  maze_set_wall(1, 2, EAST);
  maze_set_wall(1, 1, EAST);

  /*Third column*/
  maze_set_wall(2, 1, EAST);
  maze_set_wall(2, 2, EAST);
  maze_set_wall(2, 2, SOUTH);

  maze_print();
}

void loop()
{
    /*Wait for key press*/
    if(Serial.available() > 0)
    {
      /*Store key press*/
      int c = Serial.read();

      /*Run flood fill algorithm*/
      /*Move mouse*/
      /*Print maze*/

    }
}

```