Maze algorithm

A useful analogy when thinking about this algorithm is thinking about the field that is being flooded, and the water starts from one point, and it can take different paths. The principle is that there is no path that is prioritized, but instead every path gets “one step at a time”, and that way you can guarantee that you found the shortest possible path out of the labyrinth without finding all possible paths.

There are two types of data structures used in implementing this algorithm – an array and a queue. The purpose of an array is to keep track of the path that has been taken. Every element of the labyrinth (given as a matrix) has a corresponding entry in the array. The value of the element gives us information about the field that we used to come to that field. Therefore, for example 8th element in the labyrinth, has a corresponding element array[8]. If we were tracing the labyrinth, and we came to the 8th element from the 3rd element, value of array[8] would be 3. After the algorithm reached the exit, we could simply trace the elements of the array backwards, and find the fields that constitute the shortest path.

Queue is used to keep track of the next field that needs to be checked. Therefore, when you are checking a field, you need to check four neighboring fields, and if there is a field that has a value of 1 AND has not been used for a path before (you can use array for checking that, but there are a lot of other ways to do that) you enqueue the location of that element. Once you checked all of the neighboring elements, you dequeue, and repeat the same until you find the exit.

Note: There are some other ways of implementing the algorithm, but the underlying principle would remain the same.

So here is the example given in the class:

1 1 1 1 0
0 1 0 1 0
1 1 0 1 0
1 0 0 1 0
1 1 0 1 0
1 1 1 1 1

So we start at the position 0, queue is empty, and all elements of the array are -1. We can go right, so we enqueue 1, and set array[1] to be 0. Then we dequeue, and we get 1. From position 1, we can go right, so we enqueue 2, and set array[2] to be 1. We can also go down, so we enqueue 6, and set array[6] to be 1 (additional condition is that you never go back to the opening). We dequeue and we get 2. When we are at 2, we can go right, so we enqueue 3, and set array[3] to be 2. We dequeue and we get 6. When we are at 6, we can go down, so we enqueue 11, and set array[11] to be 6. We dequeue and we get 3, and so on, until one of the paths reaches the end. Once we reach the end, we backtrack the array and we see that the path is:

29 – 28 – 23 – 18 – 13 – 8 – 3 – 2 – 1 – 0