Breadth First Search in a Grid to Find Shortest Distances

Queues have many uses. One use of a queue is that it can be used to help find the fastest way out of a maze! The queue is an integral part of a search that is more generally called a Breadth First Search. The idea is as follows. Given a starting point in a maze, travel to all places adjacent to it and mark these as being 1 step away from the start point. Then, for each of these spots that are 1 step away, try travelling to all adjacent spots that are not yet visited. Mark these as 2 spots away. Continue until one of the spots you mark is an exit! By hand, it's quite easy to visually solve this problem. In order to write a program to do it for a 2D array maze, you need a queue that keeps track of each place you have reached that you haven't yet tried to visit new places from. The queue is necessary because to get shortest distances, you must process each possible location in the order in which you arrived to them. In addition, as you travel, you should mark which places you've reached so far, so that you don't try to travel to them again. This second part is critical! Without it, you'll revisit some squares many, many times, resulting in a very slow algorithm.

Let's run through an example by hand, showing what has to happen, visually in code, in both the queue and the 2D array that is storing all shortest distances. We initialize any unvisited square to -1 to indicate that it is in fact, unvisited. When we visit that square, we will update its value to the shortest distance to travel there.

Consider the following maze:

```
~~~~~~~
~XXXXX~
~X-S-X~
~--X-X~
~~~~~~~
```

Let's assume we can only move up, down, left and right from where we currently are. Let S denote the starting square, let X denote an illegal square to move to, let ~ represent the outer boundary to which we are trying to get to, and let – denote a square which is free to visit.

Our initial arrays and queue would look like this:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (2,3)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1 -1  0 -1 -1 -1
~--X-X~     -1 -1 -1 -1 -1 -1 -1
~~~~~~~     -1 -1 -1 -1 -1 -1 -1
```

At this point, we dequeue the next item and try to go to all possible adjacent squares, which are (2, 2) and (2, 4). We enqueue both of these into the queue, marking that the distance to both of them is 0 + 1 = 1. It does not matter which order we enqueue these two items. (The 0 comes from the distance to (2,3) the +1 is for the move left or right.)

After this iteration, we have the following:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (2,2), (2,4)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1 -1 -1 -1 -1 -1 -1
~~~~~~~     -1 -1 -1 -1 -1 -1 -1
```

Next, we dequeue (2,2) and look for its unvisited neighbors (note that (2,3) is visited because in the distance array, 0 is stored in that slot, not -1.) The only unvisited neighbor of (2, 2) is (3, 2), so we update the distance to (3, 2) and enqueue it to get:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (2,4), (3,2)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1 -1  2 -1 -1 -1 -1
~~~~~~~     -1 -1 -1 -1 -1 -1 -1
```

Next, we dequeue (2,4) and look for its unvisited neighbors (note that (2,3) is visited because in the distance array, 0 is stored in that slot, not -1.) The only unvisited neighbor of (2, 4) is (3, 4), so we update the distance to (3, 4) and enqueue it to get:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (3,2), (3,4)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1 -1  2 -1  2 -1 -1
~~~~~~~     -1 -1 -1 -1 -1 -1 -1
```

Now, we dequeue (3,2) and look for its unvisited neighbors The only unvisited neighbors of (3, 2) are (3, 1), and (4, 2). We update the distance to both of these and enqueue to get:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (3,2), (3,4)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1  3  2 -1  2 -1 -1
~~~~~~~     -1 -1  3 -1 -1 -1 -1
```

Theoretically, we could be smart enough to recognize that (4, 2) is on the boundary and that the shortest way out is 3. In code though, most people end up waiting to stop the search until (4, 2) is dequeued, so in this trace, we will continue from this point. Here is the current state:

```
~~~~~~~     -1 -1 -1 -1 -1 -1 -1     Queue: (3,4), (3,1), (4, 2)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1  3  2 -1  2 -1 -1
~~~~~~~     -1 -1  3 -1 -1 -1 -1
```

Now, we dequeue (3, 4) and get to this state, once we perform all necessary duties:

```
~~~~~~~~    -1 -1 -1 -1 -1 -1 -1    Queue: (3,1), (4,2), (4,4)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~     -1  3  2 -1  2 -1 -1
~~~~~~~~    -1 -1  3 -1  3 -1 -1
```

Next, we dequeue (3, 1):

```
~~~~~~~~    -1 -1 -1 -1 -1 -1 -1  Queue:(4,2),(4,4),(3,0),(4,1)
~XXXXX~     -1 -1 -1 -1 -1 -1 -1
~X-S-X~     -1 -1  1  0  1 -1 -1
~--X-X~      4  3  2 -1  2 -1 -1
~~~~~~~~    -1  4  3 -1  3 -1 -1
```

Finally, when we get here, we dequeue (4, 2) and realize that we're out of the maze and return our final answer of 3. Note: If you want, you can return this value right when this spot would have been enqueued.

In the sample problem, we'll determine the shortest distance between Peter and Cottontail, two rabbits in a grid. The general structure of the code looks like this:

---

1. Create a 2d array storing integer distances from our starting location. Initially fill this to store -1 in all slots to indicate that no shortest paths have been found.

2. Enqueue the starting location into the queue.

3. Mark the distance of the starting location to be 0.

4. While the queue isn't empty

    4a. Dequeue the next grid location. Call this cur.

    4b. Loop through each adjacent grid location (with DX/DY arrays)

        4bi. Check if the location is inbounds, has a distance of -1 (never visited) and isn't an Illegal grid square.

            4bi1. If so, add this location to the queue.

            4bi2. Mark its distance as the distance to cur plus 1 (since we got here from cur).