Lab “4”: Breadth-First Search

Due Monday December 7 at 9 AM

The absolute last day to turn this in for late credit is Friday December 11 at 11:59 PM

The breadth-first search (BFS) algorithm is used in a wide variety of applications: artificial intelligence, navigation, robotics, and many more. Today we’re going to use it to help a robot navigate a maze. Here is pseudocode for the BFS algorithm. **IMPORTANT**: continue reading this document, don’t begin coding right away. Tackling this part right away will make things more difficult than they need to be.

```
Create an empty Queue q
Mark the start location as visited and add it to q

As long as there are elements in q:
    Dequeue the next location loc from q
    If loc is the goal location, return the path by backtracing
    Otherwise, for each unvisited open neighbor location n:
        Mark n as visited
        Set n’s previous location to loc
        Add n to q

If the goal was never reached, return an empty path
```

Some of those statements are as simple as they sound -- enqueing and dequeing, setting the visited / previous location fields of your locations, etc. But some are real coding problems of their own, and for those I have given you method headers to help you out. In addition, I set up unit tests, and you should be able to see the results when you submit to make sure you’re on the right track. You can submit as many times as you want, so you can use these tests to guide your development. A few notes:

- My unit tests don’t test for absolutely everything, but they are a good indication that your implementation of these helper methods is correct
- You will have to submit all of the files (or at least Maze and Location) in order for the tests to work, because of how Mimir works (it needs to be able to compile all of your code).

Lastly, and this is so important that I didn’t even include it with the other bullet points: It’s going to be very tempting to jump straight into the BFS problem, but I promise you it will be one million times easier if you **start with the helper methods, make sure they’re correct, and then continue onto the BFS problem with the confidence that all of your pieces work correctly**. That’s why I’ve given you unit tests in the first place. It's like making sure you have all of the ingredients before you start cooking a recipe -- otherwise you might get halfway through and realize the scallions you have are 6 months old and you have to go buy more scallions before you can continue cooking, but then you realize you don't even know what scallions are.

The methods' unit tests are listed in the order that I recommend completing them:

Constructor

The constructor must initialize the private **`grid`** instance variable, which is a 2D array of **`Location`** objects. This is the **`Maze`** object's internal representation of the maze. We use the **`Location`** object to store information about whether or not that location has been visited and if it is an open space or not, as well as the actual row and column where it is located in the array. **`Location`** is already written for you with all of the accessors and mutators that I needed.

Why does the **`Location`** need to know its row and column if it's already in the grid at a particular row and column? We will be adding **`Locations`** to and removing them from the queue, at which point we won't know where in the grid they came from unless they store their own actual row and column locations.

The **`Maze`** constructor takes in a 2D array of **`char`**s which tell you which spaces should be open (represented by a .) and which should not (represented by anything but a ., like an X). Initialize **`grid`** with the same dimensions as the input array, and fill each index of **`grid`** with a new **`Location`** object with the correct row, column, and open fields (which are the parameters of the **`Location`** constructor).

This is the only time you should create new Location objects, in all other cases you'll be accessing or modifying these locations from the grid itself. If we ever mark a **`Location`** as visited, we need to be sure we're actually marking the real **`Location`** as visited, not just a copy of it with the same row/column locations.

Submit your code to make sure you pass the Constructor unit test.

getUnvisitedOpenNeighbors

This method takes in a **`Location`** and returns a **`LinkedList`** of its neighboring open **`Locations`** that have not yet been visited. So based on the row and column of the input **`Location`**, check for its neighbors in **`grid`**, adding each one to a **`LinkedList`**. We know how to find neighbors in a 2D array from past assignments (up, left, down, right (not diagonal) -- and don't go out of bounds), but this time we also need to avoid adding neighbors that have already been visited or are not open.

We'll just use Java's built-in **`LinkedList`** here. We need a **`LinkedList`** instead of an array because we don't know how many unvisited open neighbors there are when we start (there could be between 0 and 4, inclusive). To create an empty list, you can do this:

```
LinkedList<Location> neighbors = new LinkedList<Location>();
```

Then, look at each of the neighboring **`Location`**s, and decide whether or not it needs to be added to the list. For each neighbor (let's say you call the neighbor **`loc`**), you can use **`loc`**'s **`isOpen()`** and **`isVisited()`** methods to determine whether or not to add it (remember to only add neighbors that are open and unvisited -- you can go to them, but you haven't yet). Then, to add it to the list, you could say:

```
neighbors.add(loc);
```

Submit your code to make sure you pass the getUnivistedOpenNeighbors unit test.

backtracePath

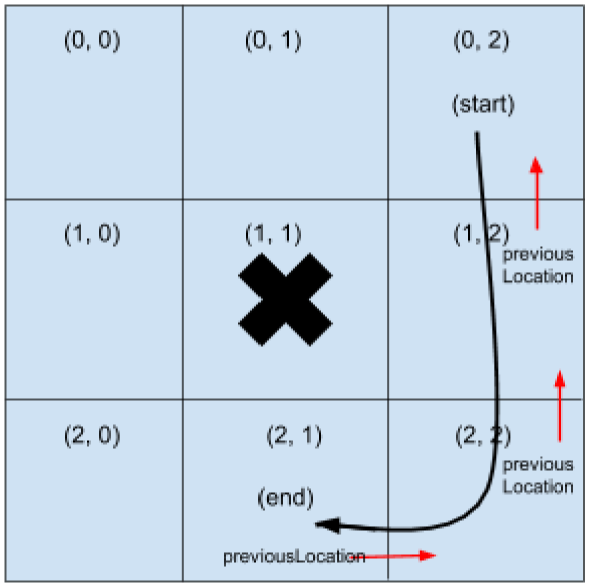
This helper method solves the following problem: Once you have located the goal **`Location`**, how do you reconstruct the path that got you there?

Assume that every location has a back-pointer, or a reference to the Node from which you came to get to this Node (you will have to set up these back-pointers in the **`shortestPath`** method, but for this helper method just assume you already have them). These will allow you to retrace your steps once you reach the goal.

For example, I have moved from Washington DC to New Hampshire, and then from New Hampshire to San Francisco. In my path, San Francisco has a back-pointer to New Hampshire, and New Hampshire has a back-pointer to Washington DC. Back-pointers make more sense than next pointers because I'm in San Francisco now -- to retrace my path, I have to start where I am and go backwards.

Each **Location** has a **`previousLocation`** field to store its back-pointer. These back-pointers form a sort of Linked List structure out of our **`Location`** objects. But instead of having a next field like a regular Node, **`Locations`** store the "next" (actually previous) **`Location`** in the **`previousLocation`** field. So, if we have the end location, we can trace the path all the way back to the start.

In this example, the **`Location`** at (2, 1) has a back-pointer to (2, 2), which has a back-pointer to (1, 2), which has a back-pointer to our start **`Location`** (0, 2). (The **`Location`** at (1, 1) is not open, which is why the path can't go through it, but that isn't relevant to this helper function.)



So, this method takes in a **`Location`** object and you must create a LinkedList (just the Java implementation, no need to create your own LinkedList class) containing the path from the start Node to the input Node (which will be the last Node in the path when you call this method later). So in the previous example, the List's first element would be the **`Location`** at index (0, 2), then the one at index (1, 2), then (2, 2), then (2, 1) -- you must put your list in the correct order, even though you are starting at the end (note that it is the reverse of the order you get when you traverse the back-pointers). Instead of using the LinkedList's **`add(T data)`** method, think about how you could use the **`add(int index, T data)`** method (or check out the Java LinkedList API for a full list of methods) to handle the ordering.

Submit your code to make sure you pass the backtracePath unit test.

And finally…

shortestPath

Now, here is the same BFS pseudocode from earlier in the PDF:

```
Create an empty Queue q
Mark the start location as visited and add it to q

As long as there are elements in q:
    Dequeue the next location loc from q
    If loc is the goal location, return the path by backtracing
    Otherwise, for each unvisited open neighbor location n:
        Mark n as visited
        Set n's previous location to loc
        Add n to q

If the goal was never reached, return an empty path
```

Now we have all of our helper methods to handle the different pieces of this algorithm! This should go without saying, but **use your helper methods to write this method**. That's why you wrote them.

Use the input parameters to get the start location from the **`grid`**, and remember not to create any new **`Locations`**. Each **`Location`** object has helpful accessors/mutators (AKA getters/setters) to deal with their various fields.

You can use the built-in Java Queue class. You can create a Queue like this:

```
Queue<Location> q = new LinkedList<Location>();
```

It looks sort of weird, but it's an example of polymorphism. The LinkedList class implements the Queue interface, so we can use the newly created q as a Queue. The Java Queue implementation has the methods **`add`** and **`remove`** instead of **`enqueue`** and **`dequeue`**, but they work just as you'd expect.

As always with pseudocode, make sure that the code you write matches up with the pseudocode. Your job is to translate this pseudocode into real code. There's no unit test for this method, you'll need to actually run it in the **`MazeDriver`**:

Running Your Code

The driver creates various 2D char arrays that you can use to try out your algorithm. Simply compiling and running the **`MazeDriver.java`** file will run your algorithm on one such maze. To try out other mazes, just follow the pattern that I used:

- Create a 2D char array filled with some pattern of open / closed spaces
- Call your **`Maze`** constructor
- Call the **`shortestPath`** method with the desired start and end coordinates
    - Optionally print out the result to make sure it's consistent with what you think the shortest path should be
- Call the **`animatePath`** method to see your algorithm in action (note that this method is already written for you). You should see your robot (represented by an O) follow the shortest path from start to finish in a text animation.

Congratulations! You just coded a robot. If SkyNet takes over, I'm blaming you.