Taylan Dillion

Graph Programming Project

Graph

I modeled the problem as a graph by setting each bug to be a node, and the web between them as edges. The initial graph is directly read in from the file and represented as a set of Bug objects, which each have an id, and a set of integers representing which bugs are adjacent to it. During the file reading process, I also needed a way to store the direction of the edge from one bug to the other. I decided to use a 2D array to do this, with the following convention: The element located on row Y, column X of the 2D array, is a string representing the direction of the web from bug with ID Y to bug with ID X. Once the Adjacency Matrix (set of Bugs) and the directions matrix (for storing directions of edges) have been built, I use the data contained in them to build a second web (Graph, Adjacency Matrix) that contains only valid possible jumps and the edges relating them. For example, in the first web, the spider could jump from bug 3 to bug 2, 14, 15, or 4. In the second (improved) web, from bug 3, the spider can only make valid moves: jumping to bug 0, 19, 30, 6. The algorithm for building the new graph is explained next.

Algorithm

The reading of the graph in from the file was simple – Line by line, if the web doesn't contain a bug with the ID of the first integer in the line, then I would create new bug, add the second integer in the line to that bug, and add that bug to the web. The third (string) in the file is stored into the directions matrix. I repeat the process for each line, starting at the second integer, to ensure that the resulting graph is undirected.

Building the second (improved) graph from the first was more difficult.

```
for (Bug a : web1)
     for (Bug b : [the neighbors of a])
          for (Bug c : [the neighbors of b])
               for (Bug d : [the neighbors of c])
                    if (a->b is the same direction as b->c and
                    b->c is the same direction as c->d)
                         // add bug a to the second web Graph)
                         // using the same process as when
                         // loading the file, and add bug d as
                         // an adjacency
```

I utilize the transitive property to guarantee that Bug a, b, c, and d are in a straight line, and thus, the spider can jump from bug a to bug d, and thus, these are added into the new graph as mutually adjacent.

Once the new graph containing only valid jumps is built, I perform a Breadth-First Search on it. This is done by giving each Bug object two new members: a Boolean visited value (we don't need three cases of visited for this graph) and an integer that represents the ID of the current bug's parent. The traceback function then starts at the final bug (the spider is considered a bug for the algorithm, with ID n + 1, 92 in this example) and visits that bug's parent, pushing each parent onto a stack, that is the solution to the maze.

Results

The program then prints the stack of solutions, and the output is as follows:

```
0, 3, 19, 44, 63, 85, 82, 59, 33, 36, 39, 42, 68, 90, 87, 84, 60,
35, 38, 41, 54, 57, 71, 52, 29, 50, 74, 77, 80, 61, 34, 37, 40, 43,
69, 66, 86, 89, 65, 45, 64, 92
```

This is the expected output, as found by solving the problem by hand.

Argument

Since it is guaranteed that the second web contains only nodes with edges pointing to where the spider can make a valid jump, the spider can never make an invalid jump in the Breadth-First Search to find node n + 1. Also, since my JUnit tests confirm that the second web is, in fact, valid (by testing the size of the second web

and the 2 nodes it should **not** contain, 13 and 14), this algorithm must solve the problem.

Bug Class

```java
package main;

import java.util.HashSet;

public class Bug {
	public HashSet<Integer> neighbors;

	public int id;

	public int parent;

	boolean visited = false;

	public Bug(int id) {
		neighbors = new HashSet<Integer>();
		this.id = id;
	}

	public void addNeighbor(int neighbor) {
		neighbors.add(neighbor);
	}
}
```

SpiderWeb Class

```
private void loadFile() throws FileNotFoundException {

		Scanner reader = new Scanner(new File(FILENAME));

		n = Integer.parseInt(reader.nextLine());

		directions = new String[n + 2][n + 2];

		for (int row = 0; row < n + 2; row++)
			for (int col = 0; col < n + 2; col++)
				directions[row][col] = "X";

		while (reader.hasNextLine()) {

			String[] line = reader.nextLine().split(" ");

			int currentBug = Integer.parseInt(line[0]);
			int adjacentBug = Integer.parseInt(line[1]);
			String dir = line[2];

			if (!hasAdded.contains(currentBug)) {
				Bug b = new Bug(currentBug);
				b.addNeighbor(adjacentBug);
				web1.add(b);
				directions[currentBug][adjacentBug] = dir;
				hasAdded.add(currentBug);
			} else {
				for (Bug b : web1)
					if (b.id == currentBug) {
						b.addNeighbor(adjacentBug);
						directions[currentBug][adjacentBug] = dir;
					}
			}
			if (!hasAdded.contains(adjacentBug)) {
				Bug b = new Bug(adjacentBug);
				b.addNeighbor(currentBug);
				web1.add(b);
				directions[adjacentBug][currentBug] =
oppositeCompassDirection(dir);
				hasAdded.add(adjacentBug);
			} else {
				for (Bug b : web1)
					if (b.id == adjacentBug) {
						b.addNeighbor(currentBug);
						directions[adjacentBug][currentBug] =
oppositeCompassDirection(dir);
					}
			}
		}
		reader.close();
	}
```

```
public void buildWeb2() {
		web2 = new HashSet<Bug>();
		hasAdded2 = new HashSet<Integer>();
		for (Bug a : web1)
			for (Bug b : getBugSetFromIntegerSet(a.neighbors))
				for (Bug c : getBugSetFromIntegerSet(b.neighbors))
					for (Bug d : getBugSetFromIntegerSet(c.neighbors))
						if (directions[a.id][b.id]
								.equals(directions[b.id][c.id])
								&& directions[b.id][c.id]

	.equals(directions[c.id][d.id])) {
							if (!hasAdded2.contains(a.id)) {
								Bug x = new Bug(a.id);
								x.addNeighbor(d.id);
								web2.add(x);
								hasAdded2.add(a.id);
							} else {
								for (Bug x : web2)
									if (x.id == a.id)
										x.addNeighbor(d.id);
							}
							if (!hasAdded2.contains(d.id)) {
								Bug x = new Bug(d.id);
								x.addNeighbor(a.id);
								web2.add(x);
								hasAdded2.add(d.id);
							} else {
								for (Bug x : web2)
									if (x.id == d.id)
										x.addNeighbor(a.id);
							}
						}
	}

public void BFS() {
		Q = new LinkedList<Integer>();
		solution = new Stack<Integer>();

		Q.add(0);
		while (Q.size() != 0) {
			int u = Q.remove();
			for (int v : getBug2(u).neighbors)
				if (!getBug2(v).visited) {
					getBug2(v).visited = true;
					getBug2(v).parent = u;
					Q.add(v);
				}
		}
	}
public void traceBack() {
		int i = n + 1;
		solution.push(i);
		while (i != 0) {
			solution.push(getBug2(i).parent);
			i = getBug2(i).parent;
		}

	}
```

Extra Credit

I used JUnit testing at each stage of my algorithm to ensure that certain cases were being met with both the original and second graph to ensure both graphs' correctness.