CSE 373, Summer 2018: Homework 7, Partner Project 4: Generating and solving mazes

Overview

In this project, you will implement Kruskal's algorithm and Dijkstra's algorithm to help you both generate and solve mazes.

You will use these files from prior assignments:

- `main.java.datastructures.concrete.dictionaries.ChainedHashDictionary.java`
- `main.java.datastructures.concrete.dictionaries.ArrayDictionary.java`
- `main.java.datastructures.concrete.ArrayHeap.java`
- `main.java.datastructures.concrete.ChainedHashSet.java`
- `main.java.datastructures.concrete.DoubleLinkedList.java`
- `main.java.misc.Searcher.java`

If you have chosen a new partner for this assignment, you will be working with your Project 3 partner again.

You will be modifying the following files:

- `main.java.datastructures.concrete.ArrayDisjointSet.java`
- `main.java.misc.graphs.Graph.java`
- `main.java.mazes.generators.maze.KruskalMazeCarver.java`

Additionally, here are a few more files that you might want to review while completing the assignment (note that this is just a starting point, not necessarily an exhaustive list):

- `main.java.datastructures.interfaces.IDisjointSet.java`
- `test.java.datastructures.TestArrayDisjointSet.java`
- `main.java.misc.graphs.Edge.java`
- `main.java.mazes.entities.Wall.java`
- `main.java.mazes.entities.Room.java`
- `main.java.mazes.entities.Maze.java`
- `main.java.mazes.gui.MainWindow.java`

This is a **team** assignment. This entire project is **due on Friday, August 10 at 11:59pm**.

When working on this assignment, we expect you meet these baseline expectations at all times:

Expectations

Here are some baseline expectations we expect you to meet:

- **You are required to pair program this assignment.** Pair programming was discussed in class, but as a reminder, this means that both you and your partner are sitting around one computer coding together at all times. One person is the "driver" who is typing code, the other person is the "navigator" who is looking on, reviewing each line of code as it is typed and checking if the code makes sense, has good logic flow, takes into account edge cases, and doesn't have typos. You and your partner are required to switch roles frequently so that by the end of the project, each partner has had equal time "driving" and "navigating".
- Follow the course collaboration policies
- **DO NOT** use any classes from `java.util.*` . There are only two exceptions to this rule:
  1. You may import and use `java.util.Iterator` and `java.util.NoSuchElementException` .
  2. You may import and use anything from `java.util.*` within your testing code.
- **DO NOT** modify instructor-provided code (unless told otherwise)
- Your program should not produce excess output, including debug-related print statements.

Part 0: Initial Setup

1. Clone your group's project from https://gitlab.cs.washington.edu/cse373-18su-project4/ and open it in Eclipse. See the instructions from project 1 if you need a reminder on how to do this (remember you must first import the project as a git project, then cancel and import as a gradle project).
2. Copy all of your data structures, algorithms, and tests from the previous project to this new one.
3. Finally, make sure everything works.

   Try running `TestGraph.java` and make sure the tests run.

Try running `SanityCheck.java`, and try running checkstyle. Checkstyle should still report the same 5 errors with `SanityCheck.java` as it did with the previous projects.

Part 1: Try running the maze generator

Task: make sure you can run `MainWindow`

Navigate to the `mazes.gui` package and run `MainWindow.java`. This will launch a program that will (eventually) generate and solve mazes.

The GUI consists of two main regions. The large region at the top, which will initially contain your maze, and a few options and buttons at the bottom.

Here is a brief explanation of the user interface:

- The "Base maze shape" combobox controls what underlying "shape" your maze will have. The default is "Grid", which creates a maze where each room is a rectangle.

  Try switching the option to "Voronoi" and click the "Generate new maze" button to the left. This will generate a maze where the rooms are more irregular.

- The "Maze generator" combobox controls the actual maze generation process – it takes the initial set of room and removes edges to generate a plausible maze.

  The default option is "Do not delete any edges", which, as you may have guessed, does not delete any edges.

  Now, try picking one of the "Delete random edges" options and generate a new maze. You should now see some of the edges removed and see something that more closely resembles a maze.

  Unfortunately, the "randomly remove edges" strategy is not that great. If we remove 30% of the edges, the maze is too easy. If we remove 50% of the edges, we often end up with an unsolvable maze. (The red dots in the upper-left and lower-right corners are our starting and ending points).

  The final option labeled "Run (randomized) Kruskal" is what you will eventually need to implement. It turns out we can use the properties of minimum spanning trees to generate a much more interesting (and yet always solvable) maze!

- The "Find shortest path" button will, once implemented, draw the shortest path between the two red dots.

  Clicking this button should currently cause the program to crash – we haven't implemented it yet.

- If you want to customize the different options (e.g. change the number of rows and columns in the grid maze, change the percentage of edges removed), look at the `MainWindow.launch` method. Feel free to change any of the constants there -- we will not be looking at this file when grading, so we don't care what changes you make.

Part 2: Implement ArrayDisjointSet

Task: complete the `ArrayDisjointSet` class.

Notes:

- Be sure to use implement your disjoint set using the array-based representation we discussed in lecture.
- We have provided you a few tests for this class (TestArrayDisjointSet), but they're deliberately minimal. We strongly encourage you to add more tests to make sure your ArrayDisjointSet is working correctly.

Part 3: Implement Graph's constructor

Task: implement the `Graph` constructor and `numVertices` and `numEdges` methods.

Notes:

- You can find this class inside the `misc.graphs` package.
- Your constructor is currently designed to accept a list of vertices and a list of edges.

  However, you may find this somewhat inconvenient to work with, especially once you start implementing the `findShortestPathBetween(...)` method.

  So, you will probably want to add extra fields so that you can store the graph in a more useful representation (e.g. as an adjacency list or adjacency matrix).
- This class uses generics in a slightly more complicated way then you've seen in previous projects. We've tried to insulate you from this as much as possible, but it's possible you may run into unexpected issues or have difficulty getting your code to compile depending on what exactly you're doing.

  If this happens to you, please ask for help either on Piazza or during office hours ASAP and we'll help you get unstuck as best as we can.

This is a class about data structures and algorithms, not a class about Java, so we don't want you to waste your time struggling with Java-related issues.

- You should make sure to look over the `misc.graphs.Edge` class. Any objects of type `E` implement the `Edge` methods.

  However, be sure not modify this class.
- You can find (fairly minimal) tests for this class in `TestGraph` .

Part 4: Implement Graph.findMinimumSpanningTree(...)

Task: implement the `Graph.findMinimumSpanningTree(...)`

Notes:

- You should implement `findMinimumSpanningTree(...)` using Kruskal's algorithm.
- You can use top `topKSort(...)` to sort your edges. This will mean your algorithm will not run in $\mathcal{O}((|E| + |V|)\alpha(|V|))$ time, but that's ok.
- If you *do* want to implement Kruskal's algorithm as efficiently as possible, feel free to implement a linear sort such as Bucket sort and use that instead.

  The only caveat is that your sorting algorithm needs to be a private helper method within your `Graph` class. It doesn't really make sense for the sorting algorithm to live in a class about graphs, but we need to do this due to limitations in our grading scripts. (They copy specific files from your projects, so if you add extra code in unexpected places, they may not be copied over.)

Part 5: Implement KruskalMazeCarver

Task: implement the `KruskalMazeCarver.returnWallsToRemove(...)` method

If you remember when we were running the maze program from before, the "remove random walls" algorithms ended up generating pretty poor mazes. They either removed too many walls and created trivial mazes, or removed too few and created impossible ones.

What we really want is an algorithm that (a) generates a random-looking maze, (b) makes sure the maze is actually solvable, and (c) removes as few walls as possible.

It turns out that we can use algorithms such as Prim and Kruskal to do exactly that!

Here's the trick: we take the maze, treat each room as a vertex and each wall as an edge, assign each wall a random weight, and run any MST-finding algorithm. We then *remove* any wall that was a part of that MST.

This will end up satisfying all three criteria! By randomizing the wall weights, we remove random walls which satisfies criteria (a). A MST-finding algorithm, by definition, will ensure there exists a path from every vertex (every room) to every other one, satisfying criteria (b). And finally, because MST-finding algorithms try and avoid cycles, we avoid removing unnecesary edges and end up with a maze where there really is only one solution, satisfying criteria (c).

Your task here is to implement this algorithm within the `KruskalMazeCarver` class.

Other notes:

- You can find this class inside the `mazes.generators.maze` package.
- Make sure you understand how to use the `Maze` and `Wall` objects. The `Wall` object is a subclass of `Edge`, which means you should be able to pass a list or set of `Wall`s into your Graph constructor (along with the corresponding list of `Room`s).
- You may import and use `java.util.Random` while implementing this class.
- If you're stuck, try taking a look at `RandomMazeCarver`, which is located within the same package. Your algorithm will be a little more complicated, but it may serve as a good source of inspiration.
- To test your method, try running the program and generate a few mazes after selecting the "Run (randomized) Kruskal" option.

Part 6: Implement Graph.findShortestPathBetween(...)

Task: implement the `Graph.findShortestPathBetween(...)` method

Finally, you will implement the code to actually solve the mazes.

Notes:

- You should implement `findShortestPathBetween(...)` using Dijkstra's algorithm.
- To represent infinity, use the `Double.POSITIVE_INFINITY` constant.

- While we definitely do encourage you to look at the pseudocode we provided in lecture, please be sure to take them with a grain of salt.

  While we tried to write the pseudocode in a way that stuck a good balance between giving you a high-level overview vs making details clear, it's possible that we accidentally omitted a few details or included a minor bug somewhere.

  In addition, you may find it convenient to arrange your code in a slightly different way then what the pseudocode suggests, especially since you're trying to solve a slightly different problem then what the lecture slides are solving: The version of Dijkstra's algorithm we provided in lecture is trying to find the shortest paths from the start to every other node; you're trying to find the shortest path from the start to a specific node.
- One challenge you may run into is figuring out how exactly to update the costs of the vertices. Instead of trying to update costs, you should add the duplicate elements into your heap, and account for this separately.
- Rather then inserting your vertices directly into your heap, you may want to create and insert a custom inner class instead. (Why do you suppose that is?)
- Once you're done and all your tests are passing, try re-running the program and click the "Find shortest path" button. If everything went well, the program should now draw a red line connecting the start and the end!

  (Or, if there is no valid path, display an alert box stating so.)

Part 7: Complete individual writeup

Task: submit answers to the following questions.

Each group member should answer and submit these questions **independently** at this canvas page.

You must submit your answers in either `.txt` or `.pdf` form.

1. **Questions about the project:**
   1. What is your name and your partner's name?
   2. How was the project? Did you enjoy it?
   3. Do you have any suggestions for improving the project?
2. **Questions about your partner:**
   1. How did you feel about pair programming this assignment
   2. How was your partnership?

Ideas for extensions

To make sure we have enough time to grade everything, we do not plan on offering extra credit on this assignment.

However, if you're bored and have some free time, something you could try exploring is implementing other algorithms for generating mazes.

In particular, Kruskal's algorithm tends to generate mazes that have lots of short "cul-de-sacs", which lowers the difficulty of the overall maze.

A better algorithm might instead try and generate longer, more "windy" paths.

This website has a good overview of a few different maze generation algorithms if you want to investigate more.

Deliverables

The following deliverables are due on **Friday, August 10 at 11:59pm**.

Code submission will be the same as project 1, by pushing and taging your code with the "SUBMIT" tag. When you and your partner are done with your code, commit and push it, then tag that commit as SUBMIT. To tag a commit in Eclipse, right-click your project and select "Team > Advanced > Tag", then write SUBMIT in the "Tag Name" field. The "Tag Message" can be whatever you want - it is like a commit message. Then click Create Tag and Start Push. It is very important that you do this instead of just "Create Tag" so that it gets pushed back to Gitlab instead of staying on your machine. If you forget to push the tag, you can do so via "Team > Remotes > Push Tags..."

Before submitting, be sure to double-check and make sure that...

- [ ] Your `ArrayDisjointSet` class is fully implemented and passes all tests.
- [ ] Your `Graph` class is fully implemented and passes all tests.
- [ ] Your `KruskalMazeCarver` class is fully implemented.
- [ ] You ran the checkstyle plugin within Eclipse and fixed all style errors.

Both partners should turn in a `.txt` or `.pdf` file containing their answers to part 7 on Canvas if you haven't already. This is due at the same time as the rest of your project.