CS 2110 Fall 2022

Assignment 7: McDiver

Table of Contents

Updates

11/21: Number of required tests for shortest-paths algorithm reduced to match comment in code.

11/18: An editing error was corrected but does not change anything about the assignment.

Learning Objectives

In this assignment you will gain experience in graph algorithms and concurrency. You will write algorithms to navigate graphs and learn how to test them. You will also learn to write concurrent code. Additionally you will have the opportunity to optimize your graph searches using various data structures like heaps and queues you've already seen. **Finally, you should stay in the habit of checking Ed, especially the pinned post "A7 FAQs". You are responsible for any clarifications made in that post.**

Introduction

The London Sewer System is well known for its complex web of underground tunnels. In this project you will model and navigate this sewer network, represented as a graph. Our protagonist, McDiver, is tasked with finding a special power ring that has been hidden in the sewer. The job is to navigate the maze to find the ring. However, this does not have to be a completely blind search. Fortunately there is a detector that can measure the distance to the ring, so the McDiver can tell when they are getting closer.

When McDiver finds the ring, it turns out that it is booby-trapped. McDiver is dropped into a lower level of tunnels where there is additional treasure sprinkled all through the maze. Fortunately, McDiver immediately finds the map to this lower level and wants to collect some treasure on the way out. But toxic fumes in the sewer are starting to take a toll. So the goal is to head to the exit in the prescribed number of steps while still picking up as much treasure as possible along the way — quickly enough to avoid falling victim to those noxious fumes!

The student submission that is the most successful at maximizing points will be recognized for excellence by the coveted *McDiver Award*.[1]

[1] No monetary prize.

Tasks To Complete

The list of concrete tasks you need to complete is given below.

Part 1:

Task 1: Implement Dijkstra's algorithm

In this task, you will complete the implementation of Dijkstra's single-source shortest-paths algorithm in the class `graph.ShortestPaths`. This implementation is needed by the program in order to generate the mazes the diver searches. In addition, you are likely to find the algorithm useful for other tasks. Of course, Dijkstra's algorithm has been covered in lecture, so the pseudocode [in the course notes] is likely to be helpful.

Note that the algorithm is to be implemented in a generic way that allows it to be reused in different parts of the program, or even in different programs. The types of vertices and nodes in the graph are parameters to the algorithm.

The algorithm uses a priority queue; we have given you a (slow) implementation of the priority queue that should be good enough for this application, but you are welcome to substitute a different one. Using a different implementation is completely optional. If you do use one, you must submit the code. It should be placed in the `datastructures` package and documented in your `summary.txt`.

Task 2: Test Dijkstra's algorithm

Having implemented Dijkstra's algorithm, you must write additional test cases to ensure that your implementation is correct. Add at least two more test cases to tests/graph/ShortestPathsTests.java. An example test case has been provided for reference.

If you are having trouble implementing Dijkstra's algorithm correctly enough to run the rest of the program, see the constant `USE_MANHATTAN_DISTANCE` in `game.Sewers.digExploreSewer()`. It offers a *temporary* workaround so you can get started on other tasks.

Task 3: Improve the use of concurrency

Once you have implemented Dijkstra's algorithm, it should be possible to run the program and see the graphical user interface. However, there is a serious inefficiency in the way the program works. When the GUI is used, the search of the maze is done in a separate concurrent thread, which must wait while the GUI thread completes its animation of the diver walking from one tile to the next. Waiting is implemented in `game.GUIControl.waitForAnimation()` as an inefficient **spin loop**. Your task is to use the concurrency primitives you have learned about to implement waiting in an efficient way. You will need to modify not only `waitForAnimation()` but also `gui.GUI.finishAnimating()`, which is called when animations are complete.

This task does not have to be completed in order to move on to the next tasks; the program should work, albeit inefficiently, with the code we give you. Be prepared to have your computer plugged in, though, because the spin loop will use 100% of your CPU until this task is complete. (You can see this in Activity Monitor or Task Manager or the equivalent, depending on your operating system.) You can expect that your computer will also run noticeably cooler when this task is complete!

Part 2:

In part 2, you will complete the following tasks in `diver/McDiver.java`:

Task 1: London Sewer System: Seek Phase

On the way to the ring (see figure below), the layout of the sewer system is unknown. McDiver does not know the full maze, but only about the place on which McDiver is standing and the immediately neighboring ones (and perhaps others that McDiver remembers). McDiver also knows the Manhattan distance to the ring. When standing on the ring, the distance is 0.

The figure below corresponds to what you will see in your GUI when running the program. In the lower left, McDiver can be seen diving into the sewer with feet sticking out. This image indicates McDiver's starting point. Currently, McDiver has traversed all the pink tiles on the way to the ring. The figure with the yellow hat is McDiver; to McDiver's east, the glowing ring can be seen.

To find the ring, McDiver will potentially have to visit all reachable points in the maze. Fortunately, the maze can be viewed as a graph, and you have seen algorithms for graph traversal! This video from Prof. Gries suggests one way to organize such a traversal (See "dfs walk").

The goal is to make it to the ring in as few steps as possible. Put your solution to this part into method `seek()` in class `diver.McDiver`. Implement it by using the methods of the SeekStatus interface to inspect the maze and to move McDiver:

> `long currentLocation()` Return the unique identifier associated with McDiver's current location.
>
> `int distanceToRing()` Return McDiver's current distance along the grid (NOT THE GRAPH) from the ring.
>
> `void moveTo(long id)` Change McDiver's current location to the node given by id.
>
> `Collection<NodeStatus> neighbors()` Return an unordered collection of NodeStatus objects associated with all direct neighbors of McDiver's current location.

Because your `seek()` method cannot see the whole maze in advance, you are unlikely to get lucky and walk the shortest possible path. However, there are heuristics you can use to improve your chances of taking a more direct route. The fewer steps you take beyond the minimum possible, the larger a score multiplier you'll receive ("Bonus" in the GUI).

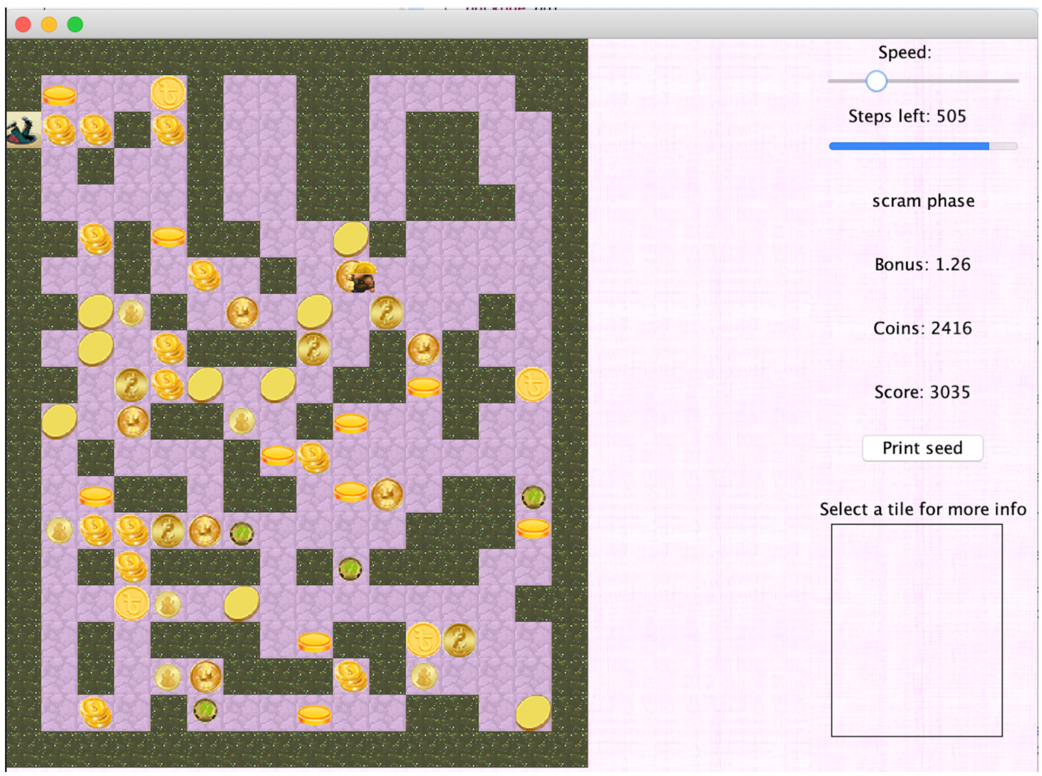
Task 2: London Sewer System: Scram Phase

Because the sewer system is an unhealthy environment, McDiver must get to the exit of the lower level within a prescribed number of steps, while also trying to pick up treasure on the way. Fortunately, McDiver now has a complete map. See Fig. 2, which also shows the pane on the right of the GUI, giving information about the sewer system. To summarize, the goal of the scram phase is to get to the exit within a prescribed number of steps, which will always be possible. In the figure below, the exit/entrance is in the upper-left corner of the sewer system—you see McDiver with feet still showing.

McDiver's score is the product of these two quantities:

1. The value of the coins that McDiver picks up during the scram phase.

2. The score multiplier from the seek phase.



Your solution to this part goes in method `scram(...)` in class `diver.McDiver`. A good starting point is to write an implementation that takes the shortest path to the exit, which is guaranteed to succeed. (Hint: use your implementation from Part 1!) After that, consider how to traverse the sewer system to pick up more coins to optimize your score using more advanced techniques. However, the most important part is always that McDiver successfully gets to the exit of the sewer system in the prescribed number of steps. If you improve on your solution, make sure that McDiver never fails to get out in time.

The scram phase is implemented by interacting with the maze and driver using the following methods from the `ScramState` interface:

`Collection<Node> allNodes()` Return a collection containing all the nodes in the graph.

`Node currentNode()` Return the Node corresponding to McDiver's location in the graph.

`Node exit()` Return the Node associated with the exit from the sewer system.

`void moveTo(Node n)` Change McDiver's location to n.

`int stepsToGo()` Return the steps remaining to get out of the sewer system.

Structure of Codebase

The structure of the code is shown below. Most of the files have helper code that you do not have to (and should not) modify. The file names colored in green have TODOs you have to complete. Files in blue have main methods that you can run.

A detailed description of each of the packages is also indicated below. You should also consult the JavaDoc documentation for the code release.

```
a7/src
├── cms.util
│    ├── maybe [dir..]
│    ├── FastException.java
├── datastructures
│    ├── PQueue.java
│    ├── SlowQueue.java
├── diver
│    ├── McDiver.java
│    ├── SewerDiver.java
├── game
│    ├── Main.java
│    ├── GameState.java
│    ├── GUIControl.java
│    ├── Edge.java
│    ├── Maze.java
│    ├── Node.java
│    ├── NodeStatus.java
│    ├── Tile.java
│    ├── ScramState.java
│    ├── SeekState.java
│    ├── Sewers.java
├── graph
│    ├── ShortestPaths.java
│    ├── DirectedGraph.java
│    ├── WeightedGraph.java
├── gui
│    ├── GUI.java
│    ├── DiverSprite.java
│    ├── MazePanel.java
│    ├── OptionsPanel.java
│    ├── Sprite.java
│    ├── SelectPanel.java
├── tests
│    ├── ShortestPathsTest.java
│
a7/res [Images needed for GUI]
```

Here is an overview of all the packages contained in this codebase.

- `cms.util:` Contains `cms.util.maybe`, which implements the Maybe type described earlier. You should use but not modify it.
- `datastructures:` This package contains implementations of data structures that can be helpful while implementing your graph search algorithms.
- `diver:` This package contains the implementations of graph traversal algorithms used in the McDiver app.
  - `McDiver.java:` You will implement the `seek()` and `scram()` phases of this game in this file
- `game:` This package contains main game state logic for the McDiver App
  - `GameState.java`: The main controller for the game.
  - `Main.java`: Runs the McDiver App. See instructions below for run configurations
- `graph:` This package contains a generic implementation of Dijkstra's shortest path algorithm that is used by the McDiver app
  - `ShortestPaths.java:` You complete the implementation for Dijkstra's algorithm in this file as part of Task 1
- `gui:` This package contains the GUI implementation for the McDiver App
- `tests:` This package contains test harnesses for various other files in this codebase
  - `ShortestPathsTest.java:` You will add your tests for Dijkstra's algorithm in this file

Running Your McDiver Program

The McDiver application can be run from the class `game.Main`.

Make sure to turn on assertions (VM option `-ea`) to get more useful feedback.

If you run the program after completing just Part 1, McDiver stands still on the screen and an error message appears telling you that `seek()` returned at the wrong location.

Some optional flags can be used to run the program in different ways.

1. *`--nographics`* runs the program in "headless" mode, with no graphical display. This is useful for evaluating performance.

2. *-n <count>* runs the program *count* number of times. This option is only available in the headless mode; it is ignored when running in GUI mode. Output is written to the console for each maze, so you know how well you did, and an average score is provided at the end. This is helpful for running your solution many times and comparing different solutions on a large number of different mazes.

3. *-s <seed>* runs the program with a predefined seed. This allows you to test your solutions on particular mazes that can be challenging or that you might be failing on. It is helpful for debugging. This can be used both with the GUI and in headless mode.

4. *--help* reports a usage message.

Scoring

Most of your points on Part 2 come from implementing a solution that always finds the ring and always gets out within the prescribed number of steps. **Your priority should be to make sure that your code always does this successfully.** If you can always find the ring and always escape, you will get at least 85% of the points for these tasks.

Beyond that basic goal, you can think of ways to optimize the score by taking fewer steps to find the ring and by collecting as much treasure as possible. Algorithms that find the ring more efficiently or that collect more treasure while escaping will get more points. Your score will not be determined by comparing to the work of other students. Full points should be achievable if your solution makes effective use of the ring distance and does a reasonable job of seeking out coins in the scram phase.

We will also care about the efficiency of your implementation. On a headless run (no GUI), it should be able to complete within 10 seconds.

Restrictions

- **You must not use any static non-final variables**. Our grading program could run several sewer systems simultaneously, using different threads, and having a static field that can change will be disastrous.
- **Import any classes and interfaces that you need from the Java API.** Don't import other things that you find on the web.

Tips

- **Don’t put all your code in the two main methods, seek and scram.** Instead, for maximum flexibility, write a dfs walk (or code to scram) in a separate method. Then, call that method from seek (or scram). This will help you manage any recursion you are using, since seek() and scram() probably should not be recursive.

- **Special maps**
  You can use the -s option to select maps in a reproducible way for the purpose of testing. Here are seed values for some interesting maps:
  A. No backtracking to find the ring (no optimization): -280019746129361794
  B. Backtracking to find the ring (no optimization): 1908492650781828577
  C. The sewer system built with this seed has no reachable coins: -3026730162232494481
  D. The only coin is on the tile with the ring, and it is picked up automatically as soon as the scram phase starts: -4004310660161599891
  E. Trivial scram: 8035820871068432943
  F. Interesting scram: 2805343804353418701

Collaboration Policy

You may do this assignment alone or with one other person. If you work with a partner, form a group on CMSX as soon as you decide to work together to formalize that agreement (if any files are submitted before confirming your group, then the group cannot be formed). Both partners must do something to form the group: one invites, the other accepts.

What to Submit

Before you submit, make sure you have made a group on CMSX with your partner! Also make sure to update the file summary.txt with information like the time you spent and your group's identification.

Then upload a zip file containing the following **six** files: `McDiver.java`, `GUIControl.java`, `GUI.java`, `ShortestPaths.java`, `ShortestPathsTest.java` and `summary.txt` to **Assignment 7** on CMSX before the deadline. The zip file should be structured like the released code, with a top-level `a7/` directory. For example, `McDiver.java` should be in `a7/src/diver` and `ShortestPathsTest.java` should be in `a7/tests/graph`. **You may also include in the zip file any extra Java source files that you added for your implementation, in the appropriate directories, and make sure that all the additional classes are under package `diver`.**

Good luck, and have fun!