CS 2110 Summer 2023

Assignment 5: Sewer navigation

Table of Contents

1. **Learning Objectives**
2. **Introduction**
3. **Structure of The Codebase**
4. **Tasks to Complete**
    a. **Part 1:**
        i. **Task 1: London Sewer System: Seek Phase (Due in Discussion)**
        ii. **Task 1: Implementing Dijkstra's algorithm**
        iii. **Task 2: Testing Dijkstra's algorithm**
    b. **Part 2:**
        i. **Task 3: London Sewer System: Scram Phase**
        ii. **Task 4: Finishing a heap implementation**
        iii. **Task 5: Optimizing your game and report your results**
5. **How to run your McDiver Program**
6. **Collaboration policy**

7. **What to Submit**

Learning Objectives

In this assignment you will gain experience with graph algorithms. You will write algorithms to navigate graphs and learn how to test them. Additionally, you will have the opportunity to optimize your graph searches using various data structures like heaps.

Just as importantly, you will practice working in a large codebase that you did not write yourself. You will need to seek out and read documentation for many classes and methods in order to determine which are relevant to your task, and you will need to juggle your role as "client" vs. "implementer" as you compose new and existing functionality to complete the project.

Timeline For Submission

- You will need to turn in a submission for Task 1 as part of your discussion on July 19th.
- Additionally, you will need to turn in a checkpoint submission for tasks 2 and 3 by the end of day Saturday July 20th.

These two submissions will not be graded for correctness, just for completion, as long as you've seriously attempted to implement these tasks, you will get full credit for this portion of the assignment.

Additionally, your final submissions and reports will be due end of day on July 28th.

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 ring with special powers that has been hidden in the sewer. His job is to navigate the maze to find the ring. Fortunately, this does not have to be a completely blind search—McDiver has a detector that can measure the distance to the ring, so he can tell when he is 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*.

Collaboration Policy

On this assignment you may work together with one partner. Having a partner is not needed to complete the assignment: it is definitely do-able by one person. Nonetheless, working with another person is useful because it gives you a chance to bounce ideas off each other and to get their help with fixing faults in your shared code. If you do intend to work with a partner, you must review the syllabus policies pertaining to partners under “programming assignments” and “academic integrity.”

As before, you may talk with others besides your partner to discuss Java syntax, debugging tips, or navigating the IntelliJ IDE, but you should refrain from discussing algorithms that might be used to solve the problems, and you must never show your in-progress or completed code to another student who is not your partner. Consulting hours are the best way to get individualized assistance at the source code level.

Frequently asked questions

If needed, there will be a pinned post on Ed where we will collect any clarifications for this assignment. Please review it before asking a new question in case your concern has already been addressed. You should also review the FAQ before submitting to see whether there are any new ideas that might help you improve your solution.

Tasks To Complete

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

Part 1:

Task 1: London Sewer System: Seek Phase

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

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 (note: the length of the path may be longer). When standing on the ring, the distance is 0.

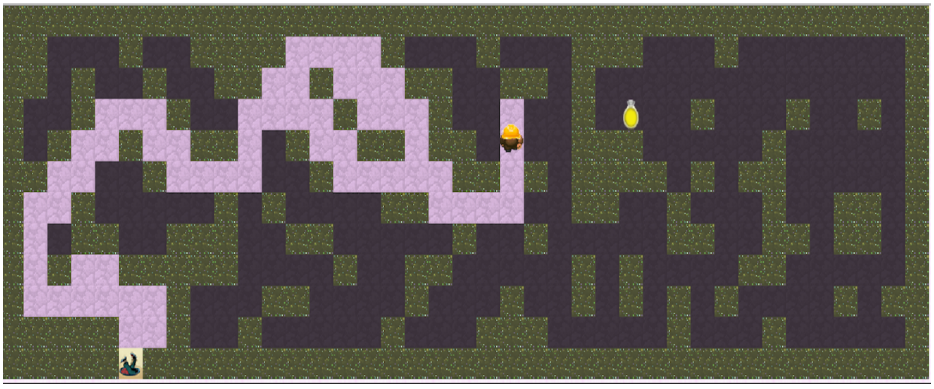
The figure below corresponds to what you will see in your GUI when running the program. In the lower left, McDiver can be seen diving into the sewer with feet sticking out. This image indicates McDiver's starting point. Currently, McDiver has traversed all the pink tiles on the way to the ring. The figure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

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, which means you can't call methods on them directly. Instead, the `WeightedDigraph` interface (and its parent, `DirectedGraph`) provide the methods you need to navigate the graph.

By reading its methods' specifications, you might notice that the `ShortestPaths` class appears to have two "lifecycle stages" - the methods `getDistance()` and `bestPath()` may not be invoked until after `singleSourceDistances()` has been run. For this reason, you as the implementer should *not* call these methods while computing shortest paths.

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 need to to substitute a different one in Task 5 after you've written your Heap implementation.

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, but it does not provide sufficient coverage (hint: the lecture example illustrates a case where a FIFO frontier was insufficient).

The A5 program requires a correct implementation of `ShortestPaths` in order to run, so if you get stuck on this method, seek assistance from consulting early!

Part 2:

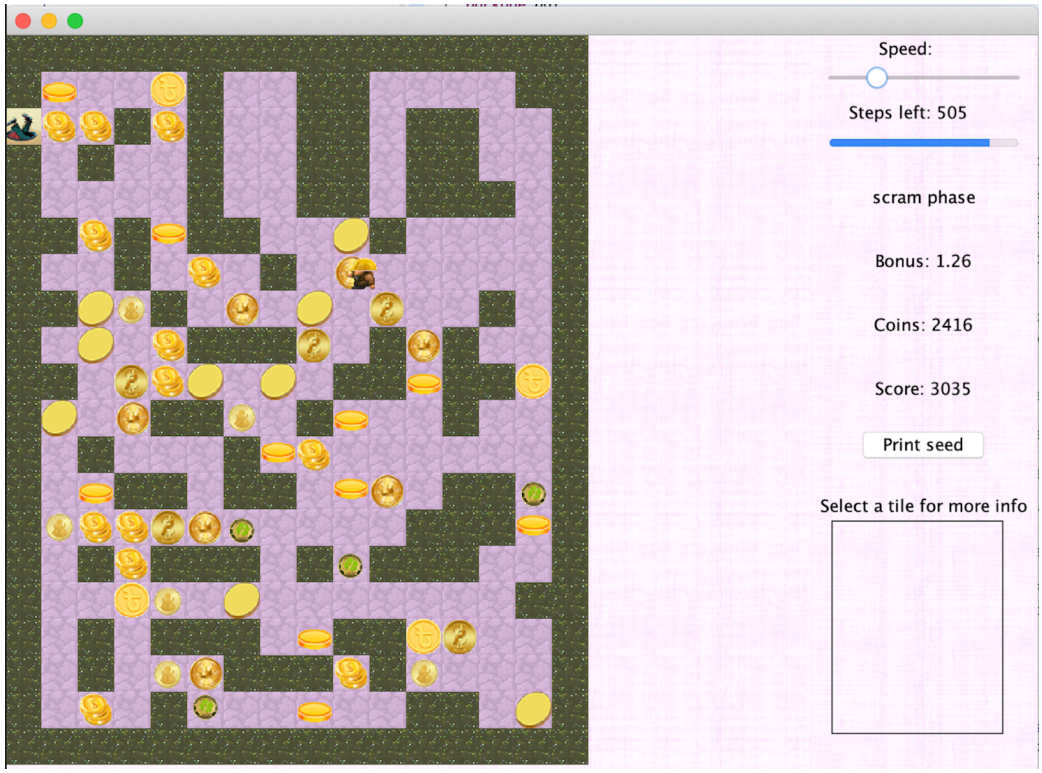
Task 3: London Sewer System: Scram Phase

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 and Requirement: 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.

If you want to use your `ShortestPaths` as part of your solution to `scram()` (highly recommended), you might find the class `game.Maze` to be useful.

Task 4: Finishing an implementation of a heap

Now finish all the TODOs in `datastructures/Heap.java` and `tests/datastructures/HeapTest.java`

Task 5: Optimizing your game and reporting your results

Before you start this task, read the rest of the document including the tips and restrictions section.

- (a) Finish writing an optimizeScram() function in `McDiver.java` function.

In this implementation of scram() you must attempt to optimize scram() in a way that increases your score for the scram phase. This is just a skeleton optimization. So don't worry about how well it's doing just yet.
[Note: that it does not matter how you achieve this. There are multiple ways of optimizing your codebase. Make sure however that your code is modular and follows the restrictions mentioned below at the end of the document]

- (b) Now you have a skeleton codebase that you can experiment with. For this portion, you and your partner should start a google doc, where you can add all your experiment results and report your findings.

Your deliverable for this task will be contained in report.pdf (save your google doc as a pdf at the end) and save that pdf in your a5/ zip folder that you will turn in on cmsx.

1. Step 1: Run your vanilla scram method (that simply uses dijkstra to navigate to the exit using the shortest path), the reference McDiver (should be found in cmx on/after Sunday July, 23rd) and the optimizeScram() method and tabulate the scores for all the seeds given in the google doc.

By this point, your google doc should have the partially filled table of the format below:

Seeds	Vanilla scram	Ref Solution	Exhaustive Scam	….
….	Coins collected : .. Bonus multiplier : … Score : ….	…	…	….
….	…	…	…	…
….	…	…	…	…

2. Step 3:
    a. Set the reportTime flag in `Main.java` to true and run Main again on all the seeds from Step 1. (It is alright with some of your run timeouts. Just run them for 4 min and report a timeout if your code doesn't finish executing in that amount of time).
    b. Now replace your Heap implementation on Dijkstra with the one you implemented in Task 5 and repeat (a). Make sure to configure your Heap correctly. **Now in your report, tabulate these results from 2.a and 2.b and discuss any improvements while using a heap on any of these implementations.**
    Note that you can always do more experiments by changing additional parameters (given in c.) to see improvements in your efficiency.
    c. In this last part, you will attempt to further optimize your scram() score from Step 2 for efficiency instead of just your score. For full points, you must successfully improve efficiency of your implementation on at least one of the following configurations below (it's fine if this comes at the cost of sacrificing some of the score you were maximizing for in (a)).
    (Hint: You may use the support of auxiliary datastructures or algorithms, but you need to implement them yourself. Do not use the data structures or algorithms in the java.util library other than the Map<> and List<> interfaces. You can use the Heap implementation you finished in Task 5. But you do not have to.)

A. sparse coins graph: Change the number of rows and columns to 400 and change density to 0.1.
B. Dense coin graph: Change the number of rows and columns to 400 and change density to 0.99

[Instructions on creating these graphs is below]

Change the number of rows and columns:

Till now, your graphs have been small enough to hopefully terminate in a reasonable amount of time. Now you will try to manually vary the configuration of the mazes to try to optimize for specific cases.
To fix the number of rows and cols to a specific number, change the following lines in game/GameState.java

```
/**
* minimum and maximum number of rows
*/
public static final int MIN_ROWS = <Desired rows>, MAX_ROWS = <Desired rows>;

/**
* minimum and maximum number of columns
*/
public static final int MIN_COLS = <Desired cols>, MAX_COLS = <Desired cols>;
```

Change the density of coins in the graph:

In order to manually vary the density, change the following line in game/Sewers.java

```
private static final double COIN_PROBABILITY = <desired density>;

// This might need to be changed for your seek to work better.
private static final double DENSITY = ...;
```

You must report clearly how you achieved your optimization and your modifications must make modifications to your scram() logic and not other parts of the codebase for this portion. Not following these guidelines and random optimizations without explaining your approach will result in significant deductions to your score.

If your code keeps timing out on graphs from above A and B even after trying to optimize your scram(). That's fine. After attempting to optimize your algorithm. Just report your findings on smaller graphs (i.e lower the rows and cols from 400 till you reach a graph size that terminates). **Note that for full credit, your optimization should show a visible efficiency improvement between your optimizeScram() in (a) and your optimization in this section on at least one graph case.**

In your report, you can try to tabulate all your trials and highlight the configuration on which you are able to achieve an improvement in efficiency when compared to (a). If you are having trouble optimizing your code for the parameters in A. and B. you can vary some other graph specific parameters (other than DENSITY) in game/Sewers.java.

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.

```
a5/src
├── cms.util
│   ├── maybe [dir..]
│   ├── FastException.java
├── datastructures
│   ├── PQueue.java
│   ├── SlowQueue.java
│   ├── Heap.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
│   ├── HeapTest.java
│
a5/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. You will need to implement a fast version of a priority queue and you can do it here.
- `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.

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 directly in the 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 themselves.

- **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:

C. No backtracking to find the ring (no optimization): -280019746129361794
D. Backtracking to find the ring (no optimization): 1908492650781828577
E. The sewer system built with this seed has no reachable coins: -3026730162232494481
F. The only coin is on the tile with the ring, and it is picked up automatically as soon as the scram phase starts: -4004310660161599891
G. Trivial scram: 8035820871068432943
H. Interesting scram: 2805343804353418701

We expect you to test your submission using these seeds to ensure that your solution is robust. There are special deductions associated with failing to run on these special maps.

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 report document with information like the time you spent and your group's identification.

Then upload a zip file named "a5.zip" containing at least the following **six** files: `McDiver.java`, `ShortestPaths.java`, Heap.java, `ShortestPathsTest.java`, `HeapTest.java` and `report.pdf` to **Assignment 5** on CMSX before the deadline. The zip file should be structured like the released code, with a top-level `a5/` directory. For example, `McDiver.java` should be in `a5/src/diver` and `ShortestPathsTest.java` should be in `a5/tests/graph`. Your report must be in the top level a5/ folder. **You may also include in the zip file any extra Java source files that you added for your implementation, in the appropriate directories**.

Good luck, and have fun!