CS 2110: Object-Oriented Programming and Data Structures

Assignment A8: Pres Pollack and the Cavern under Gates

Eric Perdew, Ryan Pindulic, Ethan Cecchetti, David Gries

April 28, 2022

1 Overview

In this assignment, you will help Cornell President Martha Pollack find the Orb of AI in a just-discovered cavern under Gates Hall. You will help her flee (get out quickly) before the cavern collapses. Also, there will be great rewards for those who help Pollack line her pockets with fallen gold on the way out. The assignment has two phases, each of which involves writing a method in Java. We explain these phases in detail later.

2 Collaboration Policy and Academic Integrity

You may complete this assignment with one other person. If you plan to work with a partner, as soon as possible —at least by the day before you submit the assignment— login to CMS and form a group. Both people must do something to form the group: one proposes and the other accepts.

If you complete this assignment with another person, you must work together as much as possible in this pandemic situation. Talk by zoom or skype or whatever is easiest for you. If one partner writes some code, the other must look at it carefully and comment on it. Both are responsible for what is written and tested.

With the exception of your CMS-registered partner, you may not look at anyone else's code, in any form. You may not look at or possess code from a similar assignment from a previous semester. You may not show or give your code to anyone else in this class, in any form. You can talk to others about the assignment at a high level, but your discussions should not include writing code or copying it down.

If you don't know where to start, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —a course instructor, a TA, a consultant. Do not wait. A little in-person help can do wonders to get you unstuck.

3 Find-the-Orb Phase

On the way to the Orb (see Fig. 1 on the next page), the layout of the cavern is unknown. Pres Pollack knows only the status of the tile on which she is standing and the immediately surrounding ones (and perhaps those that she remembers). Her goal is to make it to the Orb in as few steps as possible.

This is not a blind search, however. For each tile, the distance to the Orb is known, and this can be used greedily to optimize the find algorithm, but the greediness does not always work.

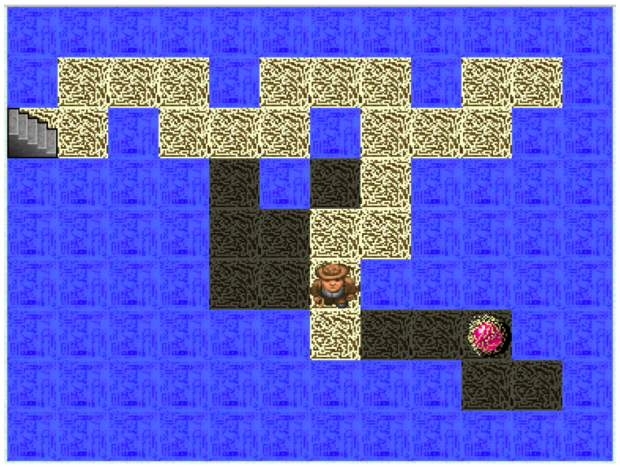


Figure 1: Finding for the Orb during the first phase

Note: In the find-the-Orb phase, all edges in the graph have weight 1.

You will develop the solution to this phase in method `find()` in class `Pollack` within package `submit`. There is no time limit for this task, but you will receive a higher score bonus multiplier for finding the Orb in fewer steps. In order to pick up the Orb, simply return. Returning when Pres Pollack is not on the Orb will throw an exception, halting the game.

Method `find()` has as parameter a `FindState` object, which contains information about the environment. Every time Pres Pollack moves, this object automatically changes to reflect her new location. This object includes the following methods:

1. `long currentLoc()`: Return a unique identifier for the tile Pres Pollack is on.
2. `int distanceToOrb()`: Return the distance from Pres Pollack's location to the Orb. This is the Manhattan distance, but you don't have to be concerned with what that term means. Look it up in JavaHyperText if you want.
3. `Collection<NodeStatus> neighbors()`: Return information about the tiles to which Pres Pollack can move from her current location.
4. `void moveTo(long id)`: Move Pres Pollack to the tile with ID id. This fails if that tile is not adjacent to her current location.

Function `neighbors()` returns a collection of `NodeStatus` objects. This object contains, for each neighbor, the ID corresponding to that neighbor, as well as the neighbor's distance to the Orb. You can examine the documentation for this class for more information on how to use `NodeStatus` objects.

We strongly suggest that you visit JavaHyperText, click on DFS/BFS in the navigation links at the top, and study the last video on a DFS walk. Study it carefully! You will find it easy to write `find()` if you understand this video.

ONCE you have a correct implementation of `find()` running, then see how you could optimize it to use the distances to the Orb of the current location's neighbors to provide some optimization. This should be a local optimization; don't redo the whole implementation and structure of your dfs walk.

4 Flee Phase

After picking up the Orb, the walls of the cavern shift and a new layout is generated. Additionally, piles of gold fall onto the ground. Luckily, underneath the Orb is a map, which reveals the full cavern. However, the stress of the moving walls has compromised the integrity of the cavern, beginning a step limit after which the ceiling will collapse. Additionally, picking up the Orb activated the traps and puzzles of the cavern, causing different edges of the graph to have different weights.

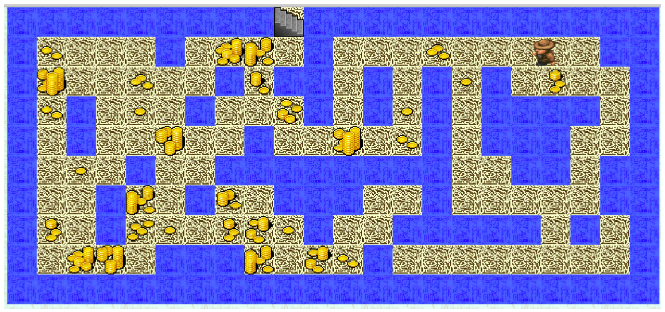


Figure 2: Collecting gold during the flee phase

The goal of the flee phase is to run to the exit from the cavern before it collapses. A score component is based on two additional factors:

1. The amount of gold that Pres Pollack picks up during the flee phase, and
2. The score multiplier from the find-the-Orb phase.

Pres Pollack's score will be the amount of gold picked up times the score multiplier from the find-the-Orb phase. Since it is fairly straightforward to simply get out of the cavern given your shortest-path implementation from A7, we expect you to spend time working to optimize your score.

You write your solution to this part in function `flee()` in class `Pollack` in package `submit`. Method `flee()` has a parameter that is a `FleeState` object, which describes Pres Pollack's environment. Every time she makes a move, this object automatically changes to reflect her new location. We describe the methods available in `FleeState`. After that, we explain a bit about the flee phase.

1. `Node currentNode()`: Return the Node corresponding to Pres Pollack's location.
2. `Node exit()`: Return the Node corresponding to the exit from the cavern (the destination).
3. `Collection<Node> allNodes()`: Return a collection of all traversable nodes in the graph.
4. `int stepsLeft()`: Return the number of steps Pres Pollack can take before the ceiling collapses.
5. `void moveTo(Node n)`: Move Pres Pollack to node `n`. This will fail if `n` is not adjacent to her location. Calling this function decrements the steps remaining by the weight of the edge from the current location to this node.
6. `void grabGold()`: Gold is picked up automatically from Pres Pollack's location. Do not call this method.

In order to get out, `return` from method `flee()` while standing on the exit. Returning while at any other position, or failing to `return` within the required number of steps, causes the application to end with a score of 0.

The cavern ceiling will collapse after Pres Pollack has taken a number of steps given by `stepsLeft()`. The steps remaining is decremented by the weight of the edge traversed when making a move, regardless of how long Pres Pollack spent deciding which move to make. You are guaranteed that Pres Pollack can get out of the cavern if she takes the shortest path out.

Whenever there is gold on Pres Pollack's location, it is automatically picked up. There is no need to call method `grabGold()`. DO NOT CALL THIS METHOD.

Classes `Node` and `Edge` have methods that you are likely familiar with from A7. Look at the documentation or code for these classes in order to learn what additional methods are available.

A good starting point is to write an implementation that will always get out the cavern within the given number of steps. From there, you can consider trying to pick up gold to optimize your score using more advanced techniques. However, the most important part is always that your solution successfully gets out of the cavern —if you improve on your solution, make sure that it never fails to escape in time.

5 What you can do

BIG SUGGESTION! Do not write much code in either `find` or `flee`. Making these methods recursive is not a good idea. Instead, write most of your code in other methods, well specified, and call those methods from `find` or `flee`. And, if you have one solution in method `A(...)` and want to begin changing it to make it better, make a copy of it, call it `B(...)`, and change `B(...)`. That way, if `B(...)` doesn't work out, you still have your copy of `A(...)` to all back on.

This is your chance to do what you want. You can write helper methods (but specify them well). You can add fields (but have comments that say what they are for, what they mean). You can change shortest-path method in class `Path` to do something a little but different. Whatever. For example, a version of it could return a HashMap containing information about the shortest paths to all nodes.

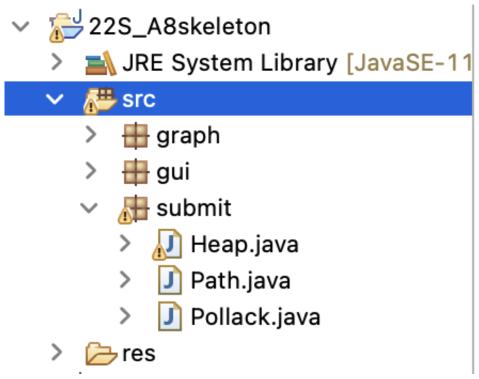
We suggest FIRST getting a solution that is simple and works. That gives you a minimum grade of 85.

Then, begin looking for ways of optimizing —always making sure you have something that works that you can submit.

6 Creating the Project in Eclipse

Download the zip file from the Assignment A8 page of our Canvas website and unzip it. It contains two directories: `src` and `res`. Start a new project for A8 and drag both directories over the project name. You will be asked whether directory `src` should be replaced. Yes it should. When you are finished, the Package Explorer should look like the figure below.

The release code contains class (`submit.Heap.java`) that we gave you for A7 as well as our solution for A7 in `submit.Path.java`. Replace out solution to `submit.Path.java` with yours if you wish.



7 Running the program

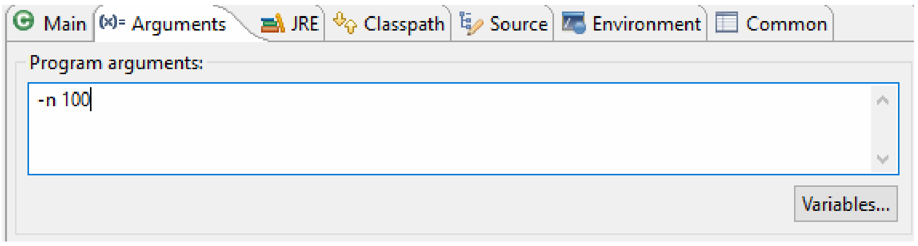
The program can be run from two classes. Running from `GameState.java` runs the program in headless mode (without a GUI); running it from `GUI.java` runs it with an accompanying display, which may be helpful for debugging. By default, each of these runs a single map on a random seed. If you run the program before any solution code is written, you will see Pres Pollack stand still and an error message pop up telling you that `find()` returned without having found the Orb.

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

1. `-n <count>` runs the program multiple times. This option is available only in headless mode; it is ignored if run with the GUI. Output will still be written to the console for each map so you know how well you did, and an average score will be provided at the end. This is helpful for running your solution many times and comparing different solutions on a large number of maps.

2. `-s <seed>`: runs the program with a predefined seed. This allows you to test your solutions on particular maps 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.

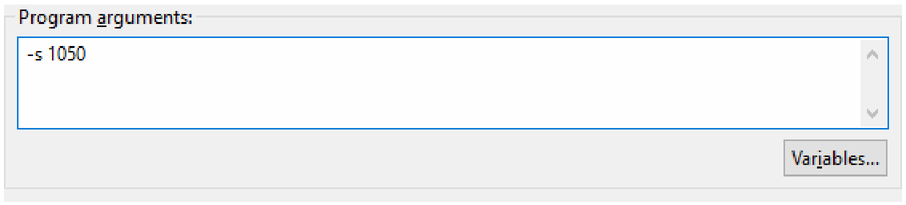
To set these arguments, in Eclipse click Run $\rightarrow$ Configurations, click on tab Arguments, and enter the arguments under Program Arguments. For instance, in order to run the program 100 times in headless mode, write:



Program arguments:

```
-n 100
```

To run the program once with a seed of 1050, write:


Program arguments:

```
-s 1050
```

When running in headless mode, you may also combine these flags. If you specify both a seed and a number of times to run, the first run will use the provided seed and subsequent runs will use additional seeds generated by the one provided. This may be useful if you want to compare two solutions on the same sequence of random maps.

8 The GUI

When running your program (except in headless mode), you are presented with a GUI where you can watch Pres Pollack making moves. When the GUI is running, each call to `moveTo()` blocks until the corresponding move completes on the GUI —that is, a call to `moveTo()` will not return and consequently your code will not continue running until the corresponding animation on the GUI has completed. For that reason, running in headless mode will generally complete faster than running it with the GUI.

The panel on the left tells you what is running: Finding the orb, Fleeing, and ScraFleem Succeeded.

You can use the slider on the left side of the GUI to increase or decrease Pres Pollacks speed. Increasing the speed will make the animation finish faster. Decreasing the speed might be useful for debugging purposes and to get a better understanding of what exactly your solution is doing. Also, you see the number of steps remaining during the flee phase. A Print Seed button allows you to print the seed to the console to easily copy and paste into the program arguments in order to retry your solution on a particularly difficult map.

You can also see the bonus multiplier and the number of coins collected, followed by the final score computed as the product of these. The multiplier begins at 1.3 and slowly decreases as Pres Pollack takes more and more steps during the find-the-Orb stage (after which it is fixed), while the number of gold coins increases as they are collected during the flee phase.

Finally, click on any square in the map to see more detailed information about it on the left, including its row and column, the type of tile, the tile's unique ID number, and the amount of gold on it.

9 Grading

A correct solution that always finds the Orb and gets out before the steps run out gets a minimum of 85/100 points! If you are pressed for time, just do that minimum. This will mean only (1) doing a dfs-walk to find the orb and (2) fleeing as quickly as possible by taking the shortest path to the exit.

To receive a higher grade, your solution must also get a reasonably high score. This is achieved by optimizing the bonus multiplier in the find-the-Orb phase and collecting as many coins as possible in the flee phase. Try for a higher grade only if you have the time and are having fun with it.

We will not look at code for this assignment. However, in the exceptional case when a submission received a very low grade, we may look at the code to see whether the grade is fair and perhaps increase the grade. But note that if extra methods are not well specified, we will not look any further. We cannot read and analyze a method if we do not know what it is supposed to do.

In the flee phase, the amount of time your code takes to decide its next move does not factor into the number of steps taken or the time remaining and consequently does not effect your score. However, we cannot wait for your code forever, so we must impose a timeout when grading your code. When run in headless mode, your code should take no longer than roughly 10 seconds to complete any single map. Solutions that take significantly longer may be treated as if they did not successfully complete and will likely receive low grades.

The use of Java Reflection mechanisms in any way is strictly forbidden and will result in significant penalties.

Points may be deducted if your submission contains `println` statements.

10 What to submit

Zip package `submit` and submit it on the CMS. Before submitting, make sure that you have not changed the interface to methods `find()` and `flee()` in class `Pollack` and that these methods work as intended. You may add other helper methods or additional classes to package `submit`, but make sure to specify everything well. In the end, make sure that if we replace our package `submit` in our solution by your package `submit` , your solution will still work as intended.

It is important that the zip file you submit contains exactly package `submit` and nothing else. To do this, select directory `submit` and do what you have to do to zip it.