CS 2110: Object-Oriented Programming and Data Structures

Assignment 8: Tennessee Morrisett and the Temple of BOOM

Eric Perdew, Ryan Pindulic, and Ethan Cecchetti

November 18, 2015

1 Overview

In this assignment, you will help avid explorer and professor of archeology Tennessee Morrisett (Dean of CIS) claim the Orb of Zot in the Temple of BOOM. You will help him explore an unknown cavern under the Temple, claim the Orb, and escape before the entrance collapses. Additionally, there will be great rewards for those who help Tennessee line his pockets with fallen gold on the way out. There are two major phases to this assignment, each of which involves writing a separate method in Java. We explain these phases in detail after presenting some further introductory material.

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 actually work together. It is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping. You should both take turns driving —using the keyboard and mouse to input code.

With the exception of your CMS-registered partner, you may not look at anyone else's code, in any form, or show your code to anyone else, in any form. You may not show or give your code to another person in the class. While you can talk to others about the assignment at a high level, 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 Exploration Phase

On the way to the Orb (see Fig. 1), the layout of the cavern is unknown. You know only the status of the tile on which you are standing and those immediately around you (and perhaps those that you remember). Your goal is to make it to the Orb in as few steps as possible.

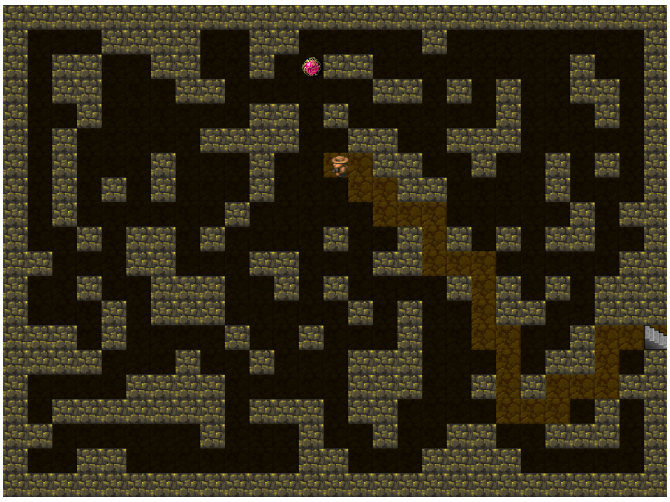


Figure 1: Searching for the Orb during the exploration phase

This is not a blind search, however. For each tile, you also know the Manhattan distance (see course Piazza note @1057) to the Orb. This is equal to the number of tiles on the shortest path to the Orb, were the explorer not impeded by walls, or equivalently:

$$|x_{\text{Orb}} - x_{\text{Explorer}}| + |y_{\text{Orb}} - y_{\text{Explorer}}|$$

Note: In the exploration phase, all edges in the graph have weight 1.

You will develop the solution to this phase in method `explore()` in `Explorer.java` within package student. There is no time limit on the number of steps you can take, 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 Tennesse is not on the Orb will cause an exception to be thrown, halting the game.

When writing method `explore()`, you are given an `ExplorationState` object to learn about the environment around you. Every time you make a move, this object automatically changes to reflect the new location of the explorer. This object includes the following methods:

1. `long getCurrentLocation()`: Return a unique identifier for the tile the explorer is currently on.
2. `int getDistanceToTarget()`: Return the Manhatten distance from the explorer's current location to the Orb.
3. `Collection<NodeStatus> getNeighbors()`: Return information about the tiles to which the explorer can move from their current location.
4. `void moveTo(long id)`: Move the explorer to the tile with ID id. This fails if that tile is not adjacent to the current location.

Note that function `getNeighbors()` returns a collection of `NodeStatus` objects. This is simply an object that contains, for each neighbor, the ID corresponding to that neighbor, as well as that neighbor's Manhattan distance from the Orb. You can examine the documentation for this class for more information on how to use `NodeStatus` objects.

A suggested first implementation that will always find the Orb, but likely won't receive a large bonus multiplier, is a depth-first search. More advanced solutions might somehow incorporate the distance of tiles from the Orb.

4 Escape 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, revealing the full cavern to you. However, the stress of the moving walls has compromised the integrity of the cavern, beginning a time 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.

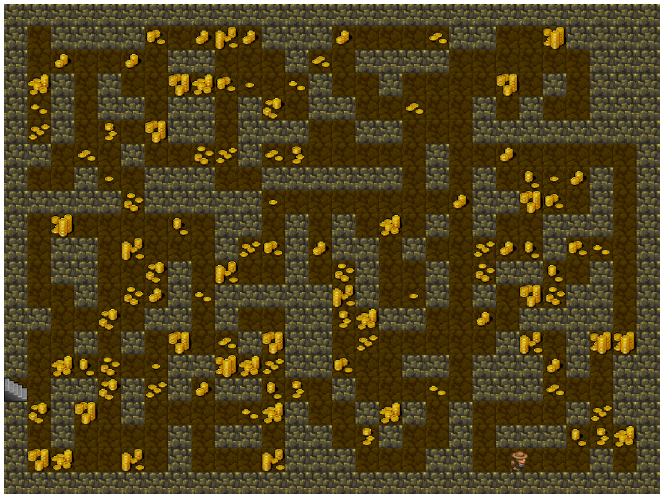


Figure 2: Collecting gold during the escape phase

The goal of the escape phase is to make it to the entrance of the cavern before it collapses. However, a score component is based on two additional factors:

1. The amount of gold that you pick up during the escape phase, and
2. Your score multiplier from the exploration phase.

Your score will simply be the amount of gold picked up times the score multiplier from the exploration phase. Since it is fairly straightforward to simply escape from the cavern given your Dijkstra's implementation from A7, we expect you to spend time working to optimize your score.

You will write your solution code to this part in function `escape()` in `Explorer.java` in package student. In order to escape, simply `return` from this method while standing on the entrance. Returning while at any other position, or failing to `return` before time runs out, will cause the game to end and result in a score of 0.

An important note is that time during this stage is not defined as the CPU time your solution takes to compute but rather the number of steps taken by the explorer: the time remaining decrements by the weight of the edge traversed when making a move regardless of how long you spent deciding which move to make. Because of this, you can be guaranteed that you will always be given enough time to escape the cavern should you take the shortest path out. Also note that there are differing amounts of gold on the different tiles. Picking up gold you are standing on takes no time.

When writing this method, you are given an `EscapeState` object to learn about the environment around you. Every time you make a move, this object will automatically change to reflect the new location of the explorer. This object includes the following methods:

1. `Node getCurrentNode()`: Return the Node corresponding to the explorer's location.
2. `Node getExit()`: Return the Node corresponding to the exit to the cavern (the destination).

3. `Collection<Node> getVertices()`: Return a collection of all traversable nodes in the graph.

4. `int getTimeRemaining()`: Return the number of steps the explorer has left before the ceiling collapses.

5. `void moveTo(Node n)`: Move the explorer to node `n`. This will fail if the given node is not adjacent to the explorer's current location. Calling this function will decrement the time remaining by the weight of the edge from the current location to this node.

6. `void pickUpGold()`: Collect all gold on the current tile. This will fail if there is no gold on the current tile or it has already been collected.

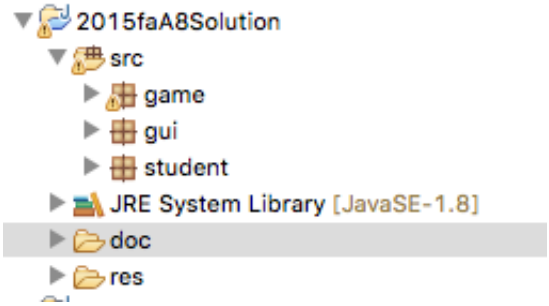
Class `Node` (and the corresponding class `Edge`) has methods that you are likely familiar with from A7. Look at the documentation or code for these classes in order to learn what additional methods are available.

A good starting point is to write an implementation that will always escape the cavern before time runs out. 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 escapes the cavern —if you improve on your solution, make sure that it never fails to escape in time.

5 Creating the Project in Eclipse

Download the zip file from the course website or the Piazza and unzip it. It contains three directories: `src`, `doc`, and `res`. Start a new project for A8 by dragging all three directories over the package name. When you are finished, the Package Explorer should look like the figure below.

The code we are giving you contains our solutions to A6 and A7: `Heap.java` and `Paths.java`. You may replace these with your own implementations —but only if you know they are correct!



6 Running the program

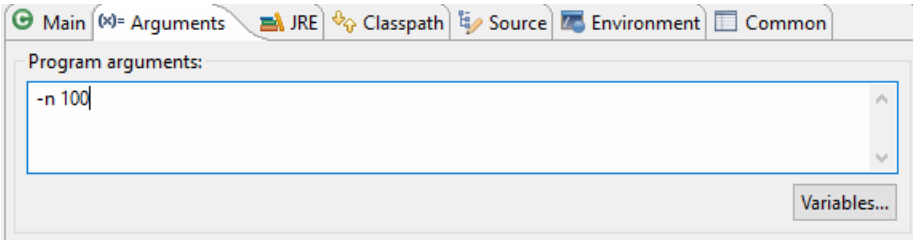
The program can be run from two classes. Running from `GameState.java` runs the program in headless mode (without a GUI); running it from `GUI.java` runs it with an accompanying display, which may be helpful for debugging. By default, each of these runs a single map on a random seed. If you run the program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

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

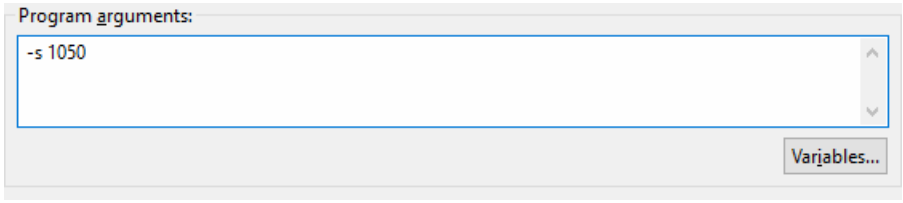
1. `-n <count>` runs the program multiple times. This option is available only in headless mode and is ignored if run with the GUI. Output will still be written to the console for each map so you know how well you did, and an average score will be provided at the end. This is helpful for running your 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 and thus is very helpful for debugging. This can be used both with the GUI and in headless mode.

To set these arguments, in eclipse click Run → 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:



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



You may combine these flags, but if you run the program more than once and provide a seed, it will run the program on the same map (the map generated by that seed) every time.

7 The GUI

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

You can use the slider on the right side of the GUI to increase or decrease the speed of the explorer. 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, a timer displays the number of steps remaining during the escape phase (both as a number and a percentage). 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 as 1.3 and slowly decreases as you take more and more steps during the explore stage (after which it is fixed), while the number of coins increases as you collect them during the escape phase.

Finally, click on any square in the map to see more detailed information about it on the right, including its row and column, the type of tile, and the amount of gold on that square.

8 Grading

The vast majority of points on this assignment will come from a correct solution that always finds the Orb and escapes before the time runs out, so your priority should be to make sure that your code always does this successfully. However, in order to receive full credit for the assignment, your solution must also get a reasonably high score (achieved by optimizing the bonus multiplier in the explore phase and collecting as many coins as possible in the escape phase), so you should spend some time thinking about ways to optimize your solution.

While 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, we cannot wait for your code forever and so 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.

9 What to submit

Zip package student and submit it on CMS. When submitting, you must make sure that you have not changed the interface to methods `explore()` or `escape()` in `Explore.java` and that these methods work as intended. You may add as many other helper methods or additional classes to package student as you want. In the end, make sure that if we replace our student package in our solution by your student package, your solution will still work as intended.

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