CS 2110: Object-Oriented Programming and Data Structures

Assignment 8: Indiana Morrisett and the Temple of BOOM

Eric Perdew, Ryan Pindulic, Ethan Cecchetti, David Gries

November 19, 2016

1 Overview

In this assignment, you will help avid explorer and professor of archeology Indiana Morrisett (Dean of CIS) claim the Orb of Zit in the Temple of BOOM. You will help him explore an unknown cavern under the Temple, claim the Orb, and get out before the cavern collapses. There will also be great rewards for those who help Indiana line his pockets with gold on the way out. The assignment has two phases, each of which involves writing a method in Java.

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. 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 Hunt-the-Orb Phase

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

This is not a blind search, however. For each tile, he also knows the Manhattan distance (see course Piazza note @1423) to the Orb, i.e. the number of tiles on the shortest path to the Orb, were Indiana not impeded by walls, or equivalently:

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

The dfs used in making Fig. 1 is naive and didn't make use of the Manhattan distance. That is why it is exploring tiles far from the orb, which is near the upper right corner.

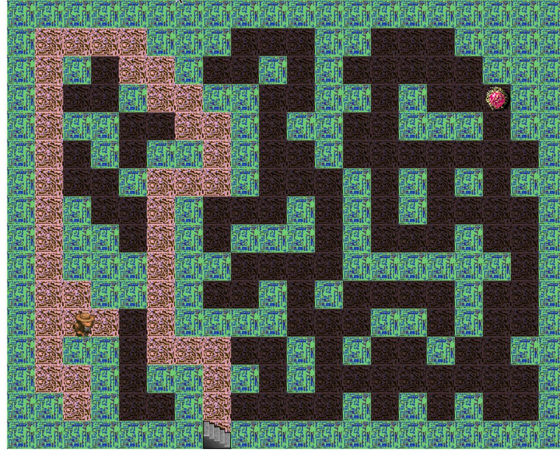


Figure 1: Searching for the Orb during the exploration phase

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

You will develop the solution to this phase in method `huntOrb()` in class `Indiana` within package `student`. 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 Indiana is not on the Orb will throw an exception, halting the game.

Method `huntOrb()` has as parameter a `HuntState` object, which contains information about the environment. Every time Indiana moves, this object automatically changes to reflect his new location. This object includes the following methods:

1. `long currentLocation()`: Return a unique identifier for the tile Indiana is on.
2. `int distanceToOrb()`: Return the Manhatten distance from Indiana's location to the Orb.
3. `Collection<NodeStatus> neighbors()`: Return information about the tiles to which Indiana can move from his current location.
4. `void moveTo(long id)`: Move Indiana to the tile with ID id. This fails if that tile is not adjacent to Indiana's 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 Manhattan distance to 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 use the distance of tiles from the Orb.

4 Scram 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. So Indiana must scram —rush to get out in time. Additionally, picking up the Orb activated the traps and puzzles of the cavern, causing different edges of the graph to have different weights. See Fig. 2. This figure shows also the pane on the left of the GUI, giving information about the cavern and its exploration.

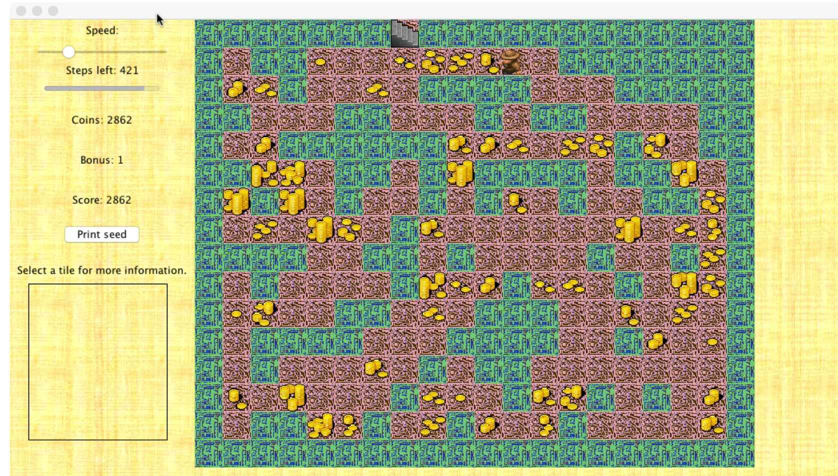


Figure 2: Collecting gold during the scram phase

The goal of the scram phase is to run to the entrance/exit from the cavern before it collapses. In Fig. 2, it's in the top row of the cavern.

Indiana's score will be the product of these two quantities:

1. The amount of gold that Indiana picks up during the scram phase.
2. The score multiplier from the Hunt-The-Orb phase.

Since it is fairly straightforward to simply get out of the cavern given your Dijkstra's implementation from A7, we expect you to spend time working to optimize your score.

You write your solution to this part in function `scram()` in class `Indiana` in package `student`. In order to get out, `return` from this method while standing on the exit. Returning while at any other position, or failing to `return` before "time" runs out, causes the game to end with a score of 0.

The cavern ceiling collapses after Indiana takes the number of steps given by function `stepsLeft()`. The steps left (i.e. remaining) is decremented by the weight of the edge traversed when making a move, regardless of how long Indiana spent deciding which move to make. You are guaranteed that there is enough time to get out of the cavern if Indiana takes the shortest path out. Also, different tiles may have different amounts of gold. Picking up gold takes no time.

Method `scram`'s parameter `state`, of type `ScramState`, describes Indiana's environment. Every time Indiana makes a move, this object automatically changes to reflect his new location. This object includes the following methods:

1. `Node currentNode()`: Return the Node corresponding to Indiana's location.
2. `Node getExit()`: 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 Indiana has left before the ceiling collapses.
5. `void moveTo(Node n)`: Move Indiana to node `n`. This will fail if `n` is not adjacent to his location. Calling this function decrements the time remaining by the weight of the edge from the current location to this node.
6. `void grabGold()`: Collect the gold on Indiana's Node (or tile). This will fail if there is no gold on the node or if 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 get out 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 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

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 Paths to do something a little different. Whatever.

We suggest FIRST getting a solution that is simple and works. That gives you a minimum grade of about 85. Save this version so you have something to submit.

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 course website. The zip file is an Eclipse project, which you can most easily import into Eclipse using menu item Open Projects From File System. Here are the steps:

1. Use menu item File $-->$ Open Project From File System

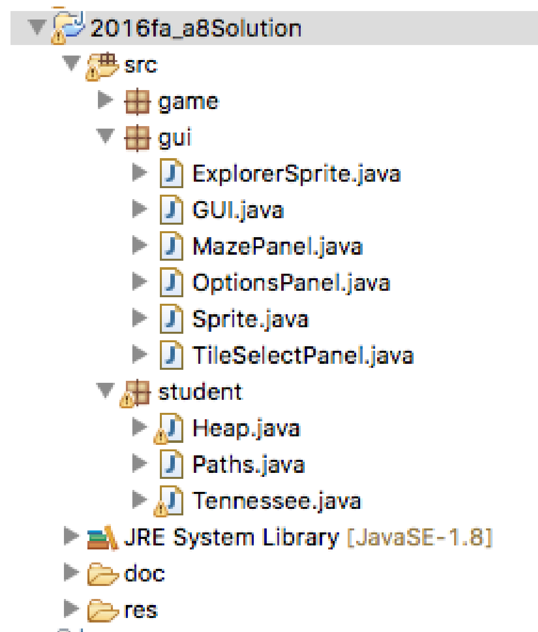
2. In the window that opens: select "archive", navigate to the downloaded zip file, select it, and click Open.

3. If you get a choice of Folders to select, select only the zip file.

4. Want to change the project name? Use the refactoring tool.

You can, if you want, copy parts individually into a project. But that will require putting JUnit4 on the build path, etc. The project should look like the diagram on the right (your JRE System Libraries may be different).

The release code contains our solution to `Heap.java`) and the skeleton for A7 (`Paths.java`). You may use your own implementations of these two classes —but only if you know they are correct. We will make our solution to A7 available on the A7 Piazza note after the deadline for A7 submission.



7 Running the program

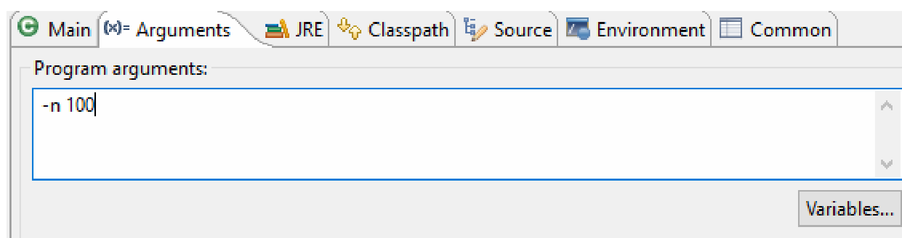
The program can be run from two classes. Running from `GameState.java` runs the program in headless mode (without a GUI); running it from `GUI.java` runs it with an accompanying display, which may be helpful for debugging. By default, each of these runs a single map on a random seed. If you run the program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

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

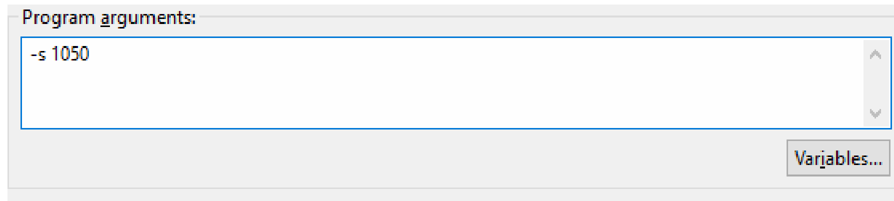
1. `-n <count>` runs the program multiple times. This option is available only in headless mode; it is ignored if run with the GUI. Output will still be written to the console for each map so you know how well you did, and an average score will be provided at the end. This is helpful for running your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



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 Indiana 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.

You can use the slider on the right side of the GUI to increase or decrease Indiana's 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, a timer displays the number of steps remaining during the get-out 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 at 1.3 and slowly decreases as Indiana takes more and more steps during the get-Orb stage (after which it is fixed), while the number of gold 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, the tile's unique ID number, and the amount of gold on that square.

9 Grading

The vast majority of points on this assignment will come from a correct solution that always finds the Orb and gets out before the time runs out, so your priority should be to make sure that your code always does this successfully. A submission that does that will receive something like a minimum of 85/100. To receive a higher grade, 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. There may also be a few bonus points available for exceptional solutions.

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.

10 What to submit

Zip package student and submit it on CMS. When submitting, you must make sure that you have not changed the signatures of methods `huntOrb()` or `getOut()` in class `Indiana` and that these methods work as intended. You may add helper methods and additional classes to package student, but make sure to specify everything well. In the end, make sure that if we replace our student package in our solution with 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.

Five (5) points will be deducted if your submission contains println statements.