Lab 4

Due: Friday 09/29/2023 @ 11:59pm EST

The purpose of labs is to practice the concepts that we learn in class. To that end you will be writing java code that uses a game engine called Sepia to develop agents that solve specific problems. In this lab we will be fighting another agent. In this game, you will be controlling some melee units (i.e. footmen units) who have a decent pool of health but don't hit very hard. Your opponent will be controlling some ranged unit (i.e. archer units) who hit pretty hard but don't have much health. Your goal is to kill the enemy before they kill you. You will be using the `Minimax` algorithm to decide what to do by thinking into the future (i.e. playing the game in your head to figure out what is the most advantagous action to choose right now).

1. Copy Files

Please, copy the files from the downloaded lab directory to your cs440 directory. You can just drag and drop them in your file explorer.

- Copy `Downloads/lab4/lib/lab4.jar` to `cs440/lib/lab4.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/lab4/data/lab4` to `cs440/data/lab4`.
  This directory contains a game configuration and map files.
- Copy `Downloads/lab4/src` to `cs444/src`.
  This directory contains our source code `.java` files.
- Copy `Downloads/lab4/lab4.srcs` to `cs440/lab4.srcs`.
  This file contains the paths to the `.java` files we are working with in this lab. Just like last lab, files like these are used to speed up the compilation process by preventing you from listing all source files you want to compile manually.
- `Downloads/lab4/doc`. While you don't have to copy this anywhere, this is the documentation generated from `lab4.jar` and will be extremely useful in this assignment.

2. Test run

If your setup is correct, you should be able to compile and execute the given template code. You should see the Sepia window appear.

```
# Mac, Linux. Run from the cs440 directory.
javac -cp lib/Sepia.jar:lib/lab4.jar:. @lab4.srcs
java -cp lib/Sepia.jar:lib/lab4.jar:. edu.cwru.sepia.Main2 data/lab4/OneVsOneInstaKill.xml

# Windows. Run from the cs440 directory.
javac -cp lib/Sepia.jar;lib/lab4.jar;. @lab4.srcs
java -cp lib/Sepia.jar;lib/lab4.jar;. edu.cwru.sepia.Main2 data/lab4/OneVsOneInstaKill.xml
```

3. Information on the provided files

- I have programmed a bit for you and generated a .jar file `cs440/lib/lab4.jar`. One recurring class, `DistanceMetric`, like usual contains some useful distance metrics that you should consider using. However, the `Vertex` class has been replaced with a `Coordinate` class, and there is no `Path` datatype. The most important section of this package is the `edu.bu.lab4.game.tree` package, as this contains a few useful datatypes when creating game trees. The primary class you will encouter is the `Node` type, which represents a single node in the game tree. Each `Node` contains a utility (from your perspective) as well as the actions you will need to immediately play in order to get to that node in the tree. Through the use of the `getChildren()` method, you can expand an entire subtree, giving you the mechanism to implement dfs algorithms such as Minimax and Alpha-Beta Pruning. Please take a look at the documention, which I have provided in this package and will also post on Piazza.

- Directory `src/lab4/agents/` contains one java file: `MinimaxAgent.java`. This agent controls a set of footmen units and should work for an arbitrary amount. The most I will test you with is a team of 2 units, as the game tree grows rather large with any team larger than a single piece. When it is your turn, you need to come up with actions for every unit you control, meaning that you cannot focus on each unit individually (i.e. you cannot run Minimax on one unit, then another independently, etc.). The good news is that the `Node` type should be able to handle teams (on both sides) that contain more than a single unit.

- `cs440/lib/Sepia.jar`. The main type in Sepia is called an `Agent`, and is the type we want to extend in order to write our own agents that can interact with Sepia. While I encourage you to become familiar with the Agent api as well as the Sepia api, for now I have taken care of most of that for you. This time however, I have left the `Agent` required methods visible to you. I strongly encourage you to read those methods and see what they do.

  - a Constructor. The `Agent` type in `Sepia` does not have a public constructor, so if you extend this class you **must** write your own. This constructor must take the player number (an `int`), and optionally may take a `String[] args` (just like a main method).
  - `initialStep(StateView, HistoryView)`. This method is called on the first turn of a game (and **only** called on the first turn of a game). We typically use it to discover quantities about the world we want to know (for instance, what are the units under my control, enemy control, etc.). It is common to set the fields of an `Agent` to some invalid state in the constructor and then to actually populate them in `initialStep`. This method returns a `Map<Integer, Action>`. Every entity in the game (unit or resource) has a unique integer id, and this return value is how your `Agent` tells each unit what to do. If you want a unit to take an action that turn, you put an entry in the map for that unit (using its id as a key) along with the action you want it to take.
  - `middleStep(StateView, HistoryView)`. This method is called every turn and should contain the main logic of your `Agent`. Just like `initialStep`, this method should return a mapping from (your) unit id(s) to the actions you want those units to take this turn.
  - `terminalStep(StateView, HistoryView)`. This method is called once (and only once) at the end of a game. To be clear, this method is called after the game has **completed**, so this method returns nothing (there is no game to perform actions in). This method is typically used for other purposes such as logging info, saving things, etc.
  - `savePlayerData(OutputStream)`. This method is used to save your `Agent` to disk. We won't make use of it very often.
  - `loadPlayerData(InputStream)`. This method, like its counterpart, is to load an `Agent` from disk. We won't make use of it very often.

More information on `Agent` methods can be found in the Sepia Agent Tutorial.

Task 1: Depth-Thresholded Minimax Algorithm with a Max Depth of 1 (50 points)

In this task, I want you to develop the minimax algorithm using a recursive strategy. Remember, Depth-Thresholded Minimax is a dfs algorithm, meaning that it lends itself well to recursion. In this task I want you do use the heuristics I provided for you, so please do **not** change them (yet). These heuristics aren't very good, but they will allow you, if minimax is implemented correctly, to kill the enemy unit (who will die in one shot) using the `data/lab4/OneVsOneInstaKill.xml` game file.

Notes

- To be confident whether or not your minimax algorithm is correct, please use the autograder. Even though your agent may kill the enemy in this task, your minimax algorithm may not be recursing correctly. The autograder will put your minimax algorithm through its paces: including looking further into the future than a single move. So, please develop your code locally until it can kill the enemy in this task, and then turn to gradescope to make sure that it is recursing correctly. Sadly, the heuristics I gave you are not good enough for the agent to kill the enemy when thinking more than one move ahead (heuristics are finnicky!). See the extra credit for fixing this.
- You may create whatever helper methods you want in order to accomplish this goal, however I am not sure that you will need to create any in `MinimaxAgent.java`.
- Minimax does not use any distance calculations itself, so please do not worry about picking a specific distance function in this task.
- When you want to test your `Agent`, please open up `data/lab4/OneVsOneInstaKill.xml` in a text editor of some kind (your machine may default to opening it in your browser, we don't want that). For this task you will want to make sure that the Minimax agent for Player 0 is defined as follows:

```
<Player Id="0">
    <AgentClass>
      <ClassName>src.lab4.agents.MinimaxAgent</ClassName>
      <Argument>0</Argument>
      <Argument>1</Argument>
    </AgentClass>
  </Player>
```

while the enemy agent for plyaer 1 is defined as follows:

```
<Player Id="1">
    <AgentClass>
      <ClassName>edu.bu.lab4.agents.ArcherAgent</ClassName>
      <Argument>1</Argument>
      <Argument>2</Argument>
    </AgentClass>
  </Player>
```

The performance of the two agents are rather sensitive to the arguments we provide them, so please don't mess with them in this task.

Task 2: Extra Credit (50 points)

Sadly, the heuristics that I gave you are not good enough for you to beat the agent when either the agent has more health, you search farther into the future (i.e. the depth threshold is larger), or both. Your extra credit is to engineer the heuristics that I gave you (you are free to make your own, get rid of the ones I created, etc.) so that you can beat stronger agents as well as beat agents by thinking farther into the future. There are two separate games that you can test your heuristics and Minimax implementation against: `data/lab4/OneVsOneFuturePlanning.xml` and `OneVsOneHealthyEnemy.xml`.
If you open `data/lab4/OneVsOneHealthyEnemy.xml`, you will find the following section for the Minimax Agent:

```
<Player Id="0">
    <AgentClass>
      <ClassName>src.lab4.agents.MinimaxAgent</ClassName>
      <Argument>0</Argument>
      <Argument>1</Argument>
    </AgentClass>
  </Player>
```

The 2nd argument, which is set to 1 in this snippet, is the depth threshold that your minimax agent uses. By changing this number you can control how far into the future minimax looks. In this game, the enemy can take multiple hits from you before dying, so since we are only looking one move in the future, this is a good place to measure the quality of your heuristics. For instance, with good heuristics, we should be able to "trap" the enemy in a corner or something and kill it: something that the current heuristics do not encode. Please use this game to improve your heuristics.
If you open `data/lab4/OneVsOneFuturePlanning.xml` you will find the following section for the Minimax Agent:

```
<Player Id="0">
    <AgentClass>
      <ClassName>src.lab4.agents.MinimaxAgent</ClassName>
      <Argument>0</Argument>
      <Argument>3</Argument>
    </AgentClass>
  </Player>
```

In this game, the minimax agent is set to look 3 moves ahead. If your minimax algorithm is correct and your heuristics are better, you should be able to kill the enemy even though you are planning much farther into the future. Feel free to play around with this value (it must be > 0) and observe how much longer it takes for the game to run the further into the future you plan. By setting this value to something like 15 or 20 you should notice a significant slowdown.

Task 3: Submitting your lab

Please submit `MinimaxAgent.java` on gradescope (just drag and drop in the file). If you tried the extra credit, please also submit your `Heuristic.java` file!