Lab 3

Due: Friday 09/22/2023 @ 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 invading enemy territory. Our unit (the red one) is tasked with infiltrating enemy territory to destroy the enemy base (called a "Townhall" in Sepia). A Townhall will appear with the letter "H" while enemy foot soldiers ("footmen" units) will appear with a "f". The enemy soldiers won't attack you if they spot you (yet) and won't retaliate once their townhall is destroyed (yet), so this is a rather survivable mission.

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/lab3/lib/lab3.jar` to `cs440/lib/lab3.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/lab3/data/lab3` to `cs440/data/lab3`.
  This directory contains a game configuration and map files.
- Copy `Downloads/lab3/src` to `cs440/src`.
  This directory contains our source code `.java` files.
- Copy `Downloads/lab3/lab3.srcs` to `cs440/lab3.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.

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/lab3.jar:. @lab3.srcs
java -cp lib/Sepia.jar:lib/lab3.jar:. edu.cwru.sepia.Main2 data/lab3/lab3.xml

# Windows. Run from the cs440 directory.
javac -cp lib/Sepia.jar;lib/lab3.jar;. @lab3.srcs
java -cp lib/Sepia.jar;lib/lab3.jar;. edu.cwru.sepia.Main2 data/lab3/lab3.xml
```

3. Information on the provided files

- I have programmed a bit for you and generated a .jar file `cs440/lib/lab3.jar`. In addition to `Vertex` and `Path` datatypes used in lab2, it contains a class named `DistanceMetric`. This class is already implemented for you but you should take a look at it if you want to use it. It implements some useful geometric distance meterics such as `euclideanDistance(Vertex a, Vertex b)`, `chebyshevDistance(Vertex a, Vertex b)`, etc.
- Directory `src/lab3/agents/` contains three .java files
  - `VanillaHillClimbingAgent.java`,
  - `StochasticHillClimbingAgent.java`,
  - `SimulatedAnnealingAgent.java`.

  While there are three agent files, I only want you to complete one of them. You are more than welcome to complete the rest for extra credit. See Task 1 below for details. Each of these three classes contains a bit of code. In particular, note the following methods of `Agent` methods required by `Sepia`.
- `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: One of the Hill Climber Agents (50 points)

The agents are roughly the same in difficutly, and have their own strengths and weaknesses.

- 9:05 lab: implement `SimulatedAnnealingAgent.java`
- 10:10 lab: implement `StochasticHillClimbingAgent.java`
- 11:15 lab: implement `VanillaHillClimbingAgent.java`

In Lab 2 we calculated the whole path in the beginning (it was called for you in `initialStep`). In this lab, however, we will be calculating where to go next at every step. For this purpose, there is one method that you **must** complete in order to be compatible with the state machine I have created as well as with the autograder: `getNextPosition` (which is called for you in `middleStep` this time). Given a `Vertex` with your position on the map and a `Vertex` with your goal, the method should calculate and return a `Vertex` of your desired next move. The coordinates of your desired next move **must** be adjacent to your current position.

Notes

- Make sure to implement the required type of hill climbing.
  For instance, the `VanillaHillClimbingAgent` should implement the vanilla hill climbing algorithm, and so on.
- You may create whatever helper methods you want in order to accomplish this goal. For instance, I would suggest, like last time, creating a method to get the neighbors of the current `Vertex`.
- If your `Agent` gets stuck, you should call `this.setIsStuck(true)` to set the appropriate field (I will be checking that this happens!).
- Please use **chebyshev** distance as your heuristic for all `Agent`(s) you implement!
- When you want to test your `Agent`, please open up `data/lab3/lab3.xml` in a text editor of some kind (your machine may default to opening it in your browser, we don't want that), you will need to look at line 3 in the following section:

```
1 <Player Id="0">
2     <AgentClass>
3         <ClassName>src.lab3.agents.StochasticHillClimbingAgent</ClassName>
4         <Argument>0</Argument>
5     </AgentClass>
6 </Player>
```

  Change the ClassName line to the `Agent` that you want to run.

```
<ClassName>src.lab3.agents.SimulatedAnnealingAgent</ClassName>
<ClassName>src.lab3.agents.StochasticHillClimbingAgent</ClassName>
<ClassName>src.lab3.agents.VanillaHillClimbingAgent</ClassName>
```

Task 2: Extra Credit (100 points)

Please complete the other two `Agent`s. For each of the remaining two `Agent`s, I will give you at most 50 extra credit points (for a total of 100 extra credit points). Please keep in mind what is said in Task 1, as all of that information also pertains to the remaining `Agent`s.

Task 3: Submitting your lab

Please submit the files that you complete on gradescope (no need to submit a directory or anything, just drag and drop the files into gradescope).