Lab 2

Due: Friday 09/15/2023 @ 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 units (the green 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" (in red) while enemy foot soldies ("footmen" units) will appear with a red "f" (our unit is a green footman unit). 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.
I have programmed a bit for you and generated a jarfile called `lab2.jar`. This code contains a few types you may find interesting such as `Vertex` and `Path` datatypes, as well as an abstract class called `MazeAgent`. 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. What you have to implement are the abstract methods (that I have left as method stubs) in the files that you need to complete. More on this later.

Task 0: Setup

Included with this file are a few things:

- `lib/lab2.jar` This file is the custom jarfile that I created for you. Please move it to your `lib` directory. More on what is contained within here later.
- `srcs.txt` This file contains the paths to both `.java` files you should complete in lab. This file (and others like it) are extremely useful for compiling more than a single java file. When compiling from the command line/terminal, rather than list all source files out, the `javac` program can accept a "manifest file" which contains all source files to compile. `javac` will read this file (one line per source file) and will compile these for us. The way we do this is rather than list all of the source files one by one, we tell `javac` that this file is a manifest file by adding the `@` annotation to it. For instance, when I compiled this code, I used the following command:

  ```
  javac -cp lib/Sepia.jar:lib/lab2.jar:.  @srcs.txt
  ```

  Note that for those of you on Windows, the command will look slightly different (`javac -cp lib/Sepia.jar;lib/lab2.jar @srcs.txt`). The difference is that Windows requires you to use a ";" rather than a ":" to separate elements of the classpath. I highly encourage you to use this file to compile your own code (and more manifest files like it in the future!)
- `data/lab2` This directory contains two different games, both of which are "mazes". Please move the `data/lab2/` directory into your own `data` directory. There is a small maze (the file that creates the world is specified by `data/lab2/SmallMazeMap.xml` while the config file to setup the agents is in `data/lab2/SmallMaze.xml`) as well as a big maze (the two files are named similarly). I encourage you to test/develop your code for the small maze *before* moving on to the big maze, as the big maze is significantly larger (you may have to resize your window in order to see all of it). In the small maze, it is just your unit and the enemy townhall, while in the big maze, there is a single enemy footman. Don't be alarmed, the enemy footman is asleep or something right now: it doesn't move. In the future though we may need to sneak around it and maybe also flee once the enemy townhall is destroyed (that would certainly make the enemy soldier angry and come looking for revenge). When you want to test your code, you can do so with the following command:

  ```
  java -cp lib/Sepia.jar:lib/lab2.jar:.  edu.cwru.sepia.Main2 data/lab2/SmallMaze.xml
  ```

and there is a similar command for playing the big maze. Reminder about the Windows specific path syntax: use ";" instead of ":" when specifying elements of the classpath.

- `src` This directory contains a few nested directories within it, but at the end of this path (`src/lab2/agents/`) you will find two files `BFSMazeAgent.java` and `DFSMazeAgent.java`. Please move the `src/lab2/` directory into your `src` directory. These two files contain the two agents that I want you to complete. As mentioned previouslyi, I have taken care of a good chunk of the sepia machinery and want you to focus instead on the two method stubs that I have provided. Both of these classes extend the custom `edu.bu.lab2.agents.MazeAgent` class contained within `lib/lab2.jar`. A `MazeAgent` just provides the machinery to translate between your code (which searches for a path from your unit's location to the enemy townhall's location) and sepia. A `MazeAgent` behaves according to a state machine: it will follow a plan (i.e. a `Stack<Vertex>`) where each vertex corresponds to a position on the map: moving from one position to the next. When the plan is exhausted, the agent expects to be **next** to the enemy townhall, at which point it will attack the enemy townhall until it is destroyed. If this state machine is broken, you will a bunch of error messages and will have to close out of the game (by clicking the "x" on the window). This state machine is exposed to you through the two methods that you need to implement. More on this in the next task.

A note on the `Path` datatype contained within `lib/lab2.jar`. A `Path` here is implemented as a reverse singly-linked list. The reason for this is twofold: to make it easier to "extend" paths, and to make comparison logic easier. One form of comparison logic in java is the `.equals(Object other)` method, which returns `true` if the `other` object is equal to `this` object. When creating a custom datatype, and you want to use it in, say a `Queue` or a `Stack`, you will need to implement this method for `contains()` to work correctly (by default `.equals(Other object)` will only check for *shallow copies* of the object, not *deep copies*). Two `Path` objects are considered equal if their *destinations* are the same, rather than the entire set of edges being equal. This is so that when you implement your methods, you can easily compare `Path`s together.
The other reason, as mentioned previously, is to make it easier to "extend" a `Path`. Creating a new `Path` here is as easy as this:

```
new Path(newDstVertex, edgeWeightFromOldDstToNewDst, oldPath)
```

where `oldPath` is the path you are trying to extend (i.e. "grow" by one edge). When doing search, we will be expanding paths a **lot**, so formulating paths like this is convenient to our needs *and* lets us use shallow copies of the shared paths (rather than deep copies so it also saves us memory).

Task 2: BFS Agent (25 points)

Please take a look at the `BFSAgent.java` file located in `src/lab2/agents`. This agent has two methods that you need to complete: `shouldReplacePlan` and `search`. The `shouldReplacePlan` method returns a `boolean`: `true` if the current plan is now invalid (for instance if something crosses your path), and `false` otherwise. The `search` method is where you should implement the BFS algorithm to find a path from the src vertex to the goal vertex. One thing to note is that your `search` method should produce a path that ends **before** the goal vertex: we actually don't want to try to occupy the enemy townhall's location, we just want to be next to it (so we can attack it). Please make use of the `Path` datatype found in the `lib/lab2.jar` file. Remember that BFS does **not** care about edge weights, so please make sure to use an edge weight of `1f`!
When you want to test your `BFSMazeAgent`, please open up the maze file that you want to run (either `data/lab2/SmallMaze.xml` or `data/lab2/BigMaze.xml`). When you open this file 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:

```
<Player Id="0">
  <AgentClass>
    <ClassName>src.lab2.agents.BFSMazeAgent</ClassName>
    <Argument>0</Argument>
  </AgentClass>
</Player>
```

The line in my example

```
<ClassName>src.lab2.agents.BFSMazeAgent</ClassName>
```

is **correct**: we want player 0 to run `src.lab2.agents.BFSMazeAgent`. If this is not what that line says, you will need to change it.

Task 3: DFS Agent (25 points)

This task is the exact same as the previous one, only please implement the DFS algorithm in `src/lab2/agents/DFSMazeAgent.java`. Remember to use an edge weight of `1f`: DFS also does not care about edge weights!
When you want to test your `DFSMazeAgent`, please open up the maze file that you want to run (either `data/lab2/SmallMaze.xml` or `data/lab2/BigMaze.xml`). When you open this file 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:

```
<Player Id="0">
  <AgentClass>
    <ClassName>src.lab2.agents.BFSMazeAgent</ClassName>
    <Argument>0</Argument>
  </AgentClass>
</Player>
```

The line in my example

```
<ClassName>src.lab2.agents.BFSMazeAgent</ClassName>
```

is **wrong**: we want player 0 to run `src.lab2.agents.DFSMazeAgent` instead of `src.lab2.agents.BFSMazeAgent`.

Task 4: Extra Credit (25 points)

I have included an extra file `src/lab2/ec/DijkstraMazeAgent.java`. Like the other two classes, you will need to implement the same methods, only this time make sure to implement Dijkstra's algorithm instead of DFS and BFS. Be sure to, and you will have to program this, consider action costs. Sepia has a `Direction` type that I have already imported for you, while the parent `MazeAgent` (of your `DijkstraMazeAgent` has a method called `getDirectionToMoveTo(Vertex src, Vertex dst)` which will return such a `Direction` (that you will need to take go to from `src` to `dst`). Be aware that this method expected `src` and `dst` to be neighbors of each other: this method will crash (on purpose) if the two vertices aren't! The `Direction` type has enum values which I want you to give individual edge weights to:

- If you need to move horizontally (`Direction.EAST` or `Direction.WEST`), please use an edge weight of `5f`.
- If you need to move down (`Direction.SOUTH`), please use an edge weight of `1f`.
- If you need to move up (`Direction.UP`), please use an edge weight of `10f`.

What we are describing is some sort of gravity, where it is easy to go down and hard to go up. When considering moving along a diagonal, please use take the two cardinal directions used in the diagonal, add the squares of their edge weights, and finally take the square root for the final edge cost. For instance, $cost(\texttt{Direction.NORTHWEST}) = \sqrt{cost(\texttt{Direction.NORTH})^2 + cost(\texttt{Direction.WEST})^2}$

When you want to test your `DijkstraMazeAgent`, please open up the maze file that you want to run (either `data/lab2/SmallMaze.xml` or `data/lab2/BigMaze.xml`). When you open this file 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.lab2.agents.BFSMazeAgent</ClassName>
4     <Argument>0</Argument>
5   </AgentClass>
6 </Player>
```

The line in my example

```
<ClassName>src.lab2.agents.BFSMazeAgent</ClassName>
```

is **wrong**: we want player 0 to run `src.lab2.ec.DijkstraMazeAgent` instead of `src.lab2.agents.BFSMazeAgent`.

Task 4: Submitting your lab

Please submit the two java files: `BFSMazeAgent.java` and `DFSMazeAgent.java` on gradescope (no need to zip up a directory or anything, just drag and drop the files). If you choose to do the extra credit, please also submit `DijkstraMazeAgent.java`.