Object-Oriented Programming and Data Structures	**Problem Set 3**
CS 2110, Spring 2014	**Due at 11:59 PM, See the CMS**
Version: 1	**Last Modified: February 26, 2014**

Overview

In A3, you will implement a boustrophedonic implementation of `learn()`. From the Greek bous meaning "ox" and strephein meaning "to turn", "boustrophedonic" literally translates to "turning as an ox in plowing". A boustrophedonic search of a map uses the pattern of an ox plowing a field: back and forth, alternating between west-to-east and east-to-west.

Installation

The following instructions walk you through the installation and setup of Danaus' source code. If you do everything correctly, Eclipse should resemble Figure 1.

1. Open a browser and navigate to the A3 page of the CMS.
2. Download `Danaus.jar` and store it somewhere safe within your file system. Likely, the download will be in a Downloads directory. This works great.
3. Open Eclipse.
4. Select menu item **File->New->Java Project**. A windowed menu will appear.
5. Enter Danaus as the project name near the top of the window.
6. Click **Finish** near the bottom of the window.
7. Select menu item **File->Import**.
8. Select **General->Archive File** and click **Next**.
9. Click **Browse...** to the right of **From archive file**
10. Navigate to `Danaus.jar` that you previously stored on your hard drive. Select it and press **OK**.
11. Click **Finish**.

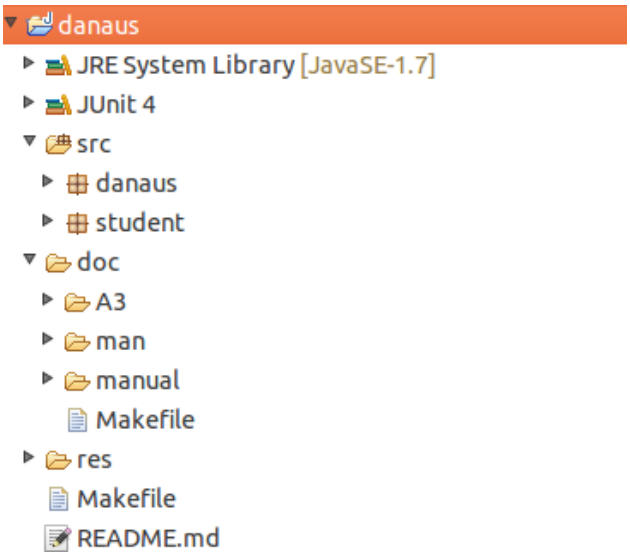


Figure 1: Eclipse after a correct installation of Danaus.

Part One: Getting Started

Before you begin implementing the boustrophedonic search, it may be helpful to play around with Danaus and acclimate to its environment. This section provides some brief suggestions of things to try with Danaus. If you already have a strong familiarity with the system, feel free to ignore this section.

Let's begin with an empty implementation of `learn()`. It will look something like this.

```
1 public TileState[][] learn() {return null;}
```

Try running Danaus with this empty learn function. A windowed GUI should pop up. The GUI should contain a graphical representation of a map and butterfly, a slider to control the frame rate of the animation, information about the state of the simulation, and information about the tile most recently clicked.

Now, let's begin flying. We can fly using function fly defined in `AbstractButterfly`. `fly` takes in two parameters, `heading` and `s`. `heading` is the direction in which you want your butterfly to fly. `s` is the speed with which the butterfly will fly. Let's attempt to fly east at a normal speed. Our learn method will now look like this.

```
1 public TileState[][] learn() {
2     fly(danaus.Direction.E, danaus.Speed.NORMAL);
3     return null;
4 }
```

Now when you run Danaus, one of two things will happen.

1. Your butterfly successfully flies east.

2. Your butterfly collides with a cliff and throws and exception.

Clearly, we do not want to prematurely terminate our program in the event of an obstacle collision. We can remedy the situation by catching the exception. Our learn method will now look like this.

```
public TileState[][] learn() {
    try {
        fly(danaus.Direction.E, danaus.Speed.NORMAL);
    }
    catch (danaus.CliffCollisionException e) {}
    return null;
}
```

Now, our learn method safely attempts to fly east. In this contrived example, we do not perform more than one fly operation, and we do not handle obstacle collisions in a meaningful way. In this assignments and in future Danaus assignments, your code will be much more complex. `RandomButterfly.java` contains a more complete demonstration of flying.

Part Two: Implement `learn()`

We assume you have read through `manual.pdf`. You can disregard content that will be introduced in later assignments. You should have a basic familiarity with Danaus, with `AbstractButterly`'s API, and with `learn()`. Note the `learn()` you implement in A3 is slightly different than the `learn()` you will implement later on in the course.

When `learn()` is called, your butterfly will be in the top left corner of the map. It is your butterfly's responsibility to traverse the map in a boustrophedonic fashion and construct a two-dimensional array of `TileStates` to represent the portion of the map your butterfly explored.

Declare a two-dimensional array of `TileStates` of the appropriate size. Get the height and width of the map using the appropriate functions in `AbstractButterfly`.

Next, have your butterfly travel east along the map using `fly()`. Continue flying until the butterfly hits either the eastern edge of the map or a cliff. Fly south one tile. Continue flying west until the butterfly once again reaches either the western edge of the map or a cliff. Once again, fly south one tile. Repeat this process until one of three terminating conditions is met.

1. Your butterfly is in the bottom row of the map flying east, and it reaches the southeast corner of the map.
2. Your butterfly is in the bottom row of the map flying west, and it reaches the southwest corner of the map.
3. Your butterfly is in the bottom row of the map flying east or west and collides with a cliff.

In A3, we guarantee that your butterfly will unconditionally be able to fly south in the event of a collision with a cliff or an edge of the map.

Every time the butterfly enters a previously unexplored tile, use `refreshState()` to retrieve its `TileState` and store it in the appropriate spot of the two-dimensional array. A tile at row `r` and column `c` should be stored in the array at row `r` and column `c`. Use `state.location.row`

and `state.location.col` to access the row and column. Once the butterfly has finished its search and the array of `TileStates` has been properly constructed, return the array. This will end the learning phase of the simulation.

Example boustrophedonic searches are given in Figure 2a, Figure 2b, and Figure 2c.

Grading

We will grade your boustrophedonic search algorithm against the course staff's implementation. The grading steps can summarized as follows.

1. Generate map.
2. Run your butterfly on the map and generate a two-dimensional array `M` of `TileStates`.
3. Run our butterfly on the map and generate a two-dimensional array `M'` of `TileStates`. Compare `M` to `M'` and generate score. If your array `M` matches `M'` exactly, you will receive a good score. If `M` is different than `M'`, you will receive a score based on the similarity.

Checklist

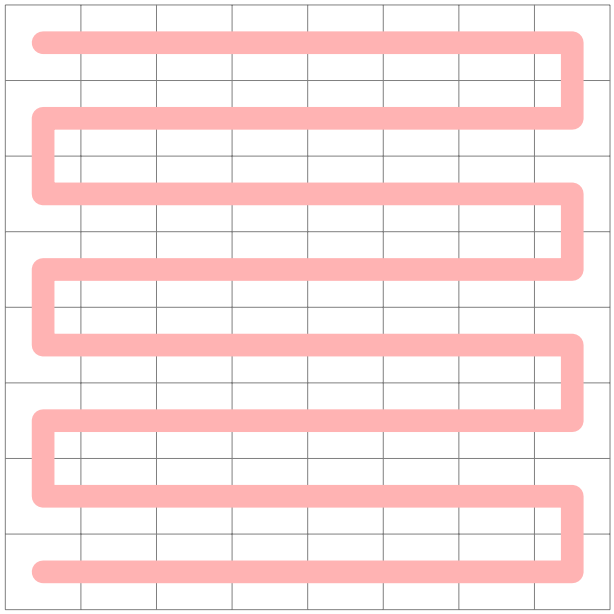
Before you submit your assignment, check to see that you have completed the following tasks.

- You have read this document and `manual.pdf` in detail.
- You have read and agreed to the academic code of integrity.
- If you have a partner, you have formed a group on the CMS. This must be done before you submit.
- You have reviewed CS 2110's coding style guidelines, and your code adheres to the guidelines.
- The fields of `Butterfly` are annotated with the class invariant.
- Each method of `Butterfly` has a good javadoc specification.
- You have placed the total time spent on the assignment at the top of `Butterfly.java`

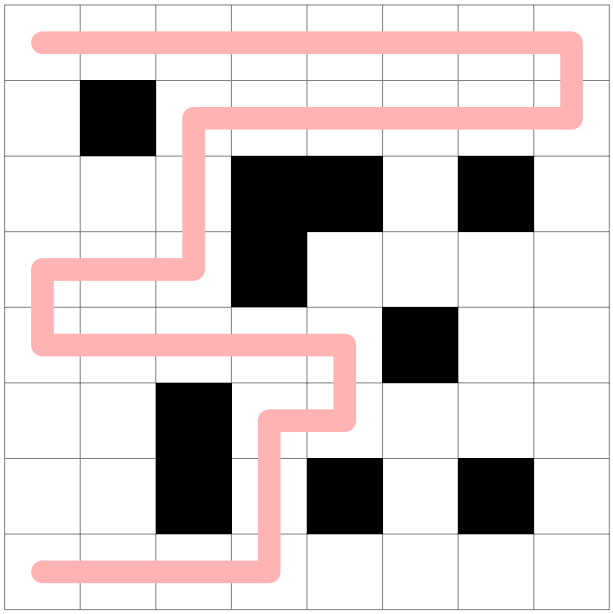
Deliverables

On the CMS, submit the following files.

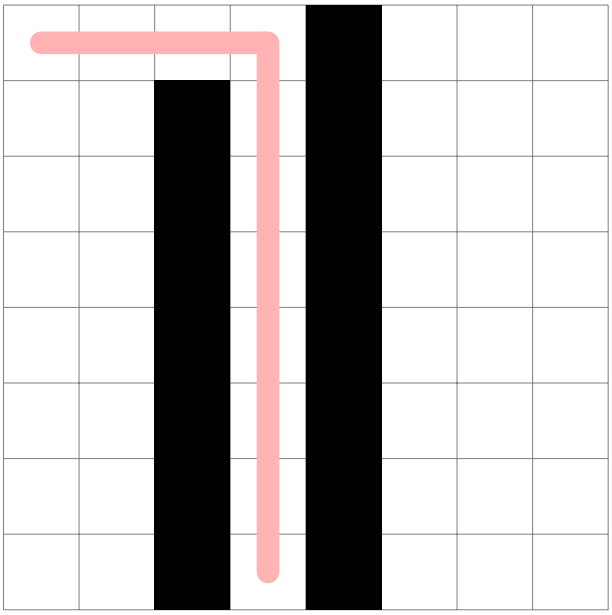
1. `Butterfly.java`. This file contains subclass Butterfly, including your implementation of a boustrophedonic `learn()`.



(a) A boustrophedonic search on a map without cliffs.



(b) A boustrophedonic search on a map populated with cliffs.



(c) A boustrophedonic search with minimal horizontal flight.

Figure 2: Sample boustrophedonic searches