Computer Science 300 – Spring 2014
Assignment #1

100 points **Due: Monday, 3/03/14 at 11:59PM**

1. (30 points) For this problem, you must solve the mapping task that we discussed in class. Recall that the robot is dropped off in a random location of a warehouse and must find all obstacles in it with as few moves as possible. The robot may only move North, West, South, or East, and can only sense the eight surrounding cells. The robot marks each empty cell that it visits and each obstacle that it senses.

   To solve this problem, you need to design and implement a simple-reflex agent with no internal state. First, download the file `Agent.zip` from D2L and unzip it to some suitable directory. All of your programming will take place in the `Robot.java`. You are not allowed to modify any other files in the zipped file that you downloaded. More specifically, you need to modify the method `pickMove` in `Robot.java`, which returns nothing but, as a side-effect, may change the direction of the robot. In this method, you will need to call the `inquire` method in the `Observer` class, as well as helper methods in the `Direction` class. Of course, you may also refer to constants in the `Layout` class. However, you are forbidden to sense the status of locations other than the eight locations adjacent to the robot. You will be severely penalized if you break this rule. Of course, you are allowed and encouraged to define helper methods and even helper inner classes. However, you should not modify any other methods in the `Robot` class. **Similarly, since the agent has no internal state, you are not allowed to add any instance or class variables to the `Robot` class.**

   The code that you downloaded implements the simple agent that was discussed in class. Compile the project, and then run the `main` method in the `Simulation` class. The program takes four command line arguments namely and in this order, the name of the layout, the display mode, the maximum number of steps that the robot is allowed to move in this simulation, and the number of times the simulation is run (in order to average the robot's performance over different starting positions and additional randomness included in your agent design, if any). For example, arguments of `"L1","g","500","100"` will run one hundred simulations for a maximum of `500` moves each in layout `L1` with the graphics display shown on the slides. The other possible values for the display mode are `t` for text/ASCII rendering, which is much faster, and `n` which is still faster since it performs no output at all. The last option is the option I will mostly use during grading.

   As your agent will be tested based on how many obstacles are discovered in a specified time duration, make sure you test it extensively since it needs to perform well on average for different starting positions in a given layout as well as in a variety of layouts. The directory that you downloaded contains five sample layout files `L1` through `L5`.

You will submit the `Robot.java` file and a summary (in your `Readme` file described later) of your experiments in the form of a table that describes your data for each of the 5 maps over `100` simulations, with `500` moves per simulation, as follows:

```
Repeat 5 times, changing the mapname each time:
   <mapname>,<average obstacles found over all simulations>  (1 row)
```

2. (70 points) Download the file `A1Part2.zip` from D2L. It contains the files needed for this part of the assignment. A rudimentary framework for search algorithms is implemented in the files that you downloaded. These files consist of:
   - `State.java` - The class that defines a state in any of our problems. This is a very basic class that stores a String and an integer. You can extend it if you need more advanced states.
   - `Node.java` - This is an abstract class that encapsulates the state and stores other book-keeping values. You will need to extend this class if you define your own problems.
   - `Problem.java` - This is the base, abstract class that describes a particular problem. You will need to extend this class if you define your own problems.
   - `SearchAlgorithm.java` - This is the base, abstract class for all our search algorithms. You will need to extend this class when you implement your own search algorithms.
   - `Solver.java` - This driver class that matches a search algorithm with a problem and then runs the set and prints out the solution. You will need to modify this class when you implement new algorithms and add additional problems.

   The inputs to the framework are given as command line arguments.

   Study these classes and their API's carefully. I will briefly go over the API in class and show you how to use them. To give you further practice with using this framework, two example problems and two algorithms have been provided. These are:
   - The Romania Map described in the lecture notes, in the files `RomaniaProblem.java` and `RomaniaNode.java`
   - The Missionaries-Cannibals problem, a variant of the jealous husbands problem presented in class, in the files `MCProblem.java` and `MCNode.java`
   - The Breadth-First-Search with Duplicate Detection (BFSDD) algorithm in the file `BFS_DD.java` . This uses a linked list queue for the open list and a hashset for the closed list.
   - The Uniform-Cost-Search with Duplicate Detection (UCS) algorithm in the file `UCS.java` . This uses a priority queue for the open list and a hashset for the closed list.

   Now provide answers to the following questions:

   a) (40 points) **State Space Definition**: Assume you are given two shot glasses, `S1` with capacity `3 oz` and `S2` with capacity `4 oz`. Both glasses are initially empty. The goal is to have exactly `2 oz` of vodka in `S2`, with `S1` empty. Also assume we are really picky about getting this `2 oz`, have access to a considerable amount of vodka and have no qualms about wasting a good amount of it (by simply pouring it away). Implement the Problem and Node files for the Shot glass problem, naming them `ShotGlassProblem` and `ShotGlassNode` respectively. To test your implementation, run your problem space using the `BFSDD` algorithm provided. For hints on how to proceed, look at the implementation of the `MC` problem file.

b) (30 points) **IDS Implementation**: Using the BFSDD algorithm as a model, implement the iterative deepening search algorithm in a class called `IDS` . Since IDS uses DFS, you will need to have a DFS method in the IDS algorithm class. Use the threshold value in the `SearchAlgorithm` class to control the maximum depth but leave it at the value currently set. Test your implementation on the `MCProblem` to verify it works.

Submission

Submit the following files, zipped up in a single folder called `Assignment1.zip` to the `Assignment1` drop box on D2L by the deadline specified:

1) A `Readme` file with the following information in it:
   a) Question 1:
      - Successfully completed: Yes/No/Yes but...
      - 5 to 10 lines that provide a brief description in English of the strategy used by your agent
      - The table of your results, in the format specified in the question.
   b) Question 2(a):
      - Successfully completed: Yes/No/Yes but...
   c) Question 2(b):
      - Successfully completed: Yes/No/Yes but...
2) `Robot.java` for Question 1.
3) `ShotGlassProblem.java` and `ShotGlassNode.java` for Question 2(a).
4) `IDS.java` for Question 2(b).