INTRODUCTION TO PROGRAMMING
DM550, DM857, DS801 (Fall 2020)
Exam project: Part II — Deadline: 23h59 on Friday, November 27th, 2020

Overview

In this phase of the project, your task is to develop the low-level classes at the lowest layers of the ant colony simulation – classes `Node`, `Edge` and `Ant` – as well as the parts of the simulation that were missing from the first phase of the project – class `Simulator`.

Class `Node`

Instances of this class represent possible locations where ants are located, which store (possibly empty) reserves of sugar. This class provides the following methods:

- a constructor with no arguments that creates a new node with no sugar;
- a constructor with one argument that creates a new node with the given amount of sugar;
- a method `int sugar()` that returns the amount of sugar in this node.
- a method `void decreaseSugar()` that decreases the amount of sugar in this node by one unit (there must be sugar in this node);
- a method `void setSugar(int sugar)` that resets the amount of sugar in this node to a given value;

Class `Edge`

Instances of this class represent possible paths between `Node`s that ants can traverse. Each `Edge` is assigned a level of pheromones that changes over time. This class provides the following methods:

- a constructor that takes to `Node`s and creates a new edge between them with no pheromones;
- methods `Node source()` and `Node target()` that return references to the source and target of this edge, respectively;
- a method `int pheromones()` that returns the level of pheromones in this edge;
- a method `void decreasePheromones()` that decreases the level of pheromoes in this edge by one unit;
- a method `void raisePheromones(int amount)` that increases the level of pheromones in this edge by the given amount.

Edges are understood to be symmetric, so that ants can travel on them in any direction. The usage of the names "source" and "target" reflects common practice in graph theory, and not their usual semantics in natural language: in our simulation, an ant can go from an edge's target to its source.

Class `Ant`

Instances of this class represent individual ants, which are placed in the graph, belong to a `Colony`, and may be carrying a fixed amount of sugar. This class should provide the following methods:

- a constructor that takes as argument a `Colony` and creates a new ant belonging to the given colony;
- a method `void move(Node location)` that moves this ant to the specified location;
- a method `Node current()` that returns this ant's current location;
- a method `Node previous()` that returns this ant's previous location (i.e., before the last invocation of `move()` – this may be the same as returned by `current()`);
- a method `Colony home()` that returns the colony that this ant belongs to;
- methods `boolean isAtHome()` and `boolean wasAtHome()` indicating respectively whether this ant's current (previous) location coincides with the colony it belongs to;

- a method `boolean carrying()` that indicates whether this ant is currently carrying sugar;
- a method `void pickUpSugar()` that models this ant picking up a fixed amount of sugar from its current location;
- a method `void dropSugar()` that models this ant dropping off its sugar load at the colony it belongs to.

There is some redundancy in this class, as some methods offer functionality that can also be achieved by simple combinations of other methods. This is to make it easier for clients to use the class, in particular avoiding the need for type casts between `Node` and `Colony`.

Class `Simulator`

Instances of this class correspond to states of a simulation, with a fixed graph, a set of ants positioned on nodes on that graph, and information about the simulation parameters. This class should provide the following methods:

- a constructor that takes as parameters a `Graph`, an array of `Ants`, the amount of sugar carried by an ant and the amount of pheromones dropped by an ant, and creates a new simulation;
- a method `void tick()` that updates the simulation by one time unit.

For compatibility with the visualizer, it is important that the parameters provided to the constructor be stored directly in attributes (and not copied), as they are shared with the instance of `Visualizer` created by the client class.

Method `tick()` does the majority of the work in the simulation. First, the pheromone levels in all edges of the graph are decreased by one unit. Then, the method goes through all the ants and update their status. Ants that arrived home in the previous tick must eat some sugar, otherwise they die; the amount of sugar eaten by an ant is 1 unit. All other ants make a move on the graph.

The moves are decided as follows.

- If an ant not carrying sugar is at a node that contains sugar, then the ant picks up sugar from the node and returns to the node it came from. For simplicity, the sugar level at a node is measured relative to the amount an ant carries – so the sugar level at the node is decreased by a unit.
- If there is only one neighbour from the ant's node, then the ant moves to that node.
- Otherwise, the ant will move to a random node that is not the node it came from. The probability that it will move from its current node n_a to another node n is given by

$$\frac{pheromones(n_a, n) + 1}{\left(\sum_{i=1}^{k} pheromones(n_a, n_i)\right) + k},$$

  where $n_1, \ldots, n_k$ are the possible nodes the ant may move to (including n) and $pheromones(n_a, n_i)$ denotes the level of pheromones in the edge between n_a and n_i. The level of pheromones in the edge travelled by the ant is increased by the amount given as parameter to the constructor.

If the ant arrives at its colony and it is carrying sugar, then it drops its sugar at the colony, and the colony's stock is increased by the amount given as parameter to the constructor.

Important notes.

- Due to the way that the visualizer is implemented, if an ant is dead then its corresponding pointer in the array of ants in the instance of `Simulator` must be set to `null`.
- Ants that are at home and do not move still need their position to be updated, in order to get the correct information in the next tick.
- Given the complexity of the moves, it might be a good idea to create a private method for performing the task of simulating a move by a given ant.
- Check the available methods in the provider classes used, namely class `Graph`.

Expected results

You must hand in a zip file containing no subdirectories and the following files.

1. Java source files `Node.java`, `Edge.java` and `Ant.java` and `Simulator` implementing the contracts described above. The first three classes should be providers for the remaining classes, which are provided in compiled form, while `Simulator` is a provider for `RunSimulator` and a client of all others. An implementation of class `RunSimulator` is also provided. Your classes should be compatible with it, as well as with your original implementation developed in the first part of the project.

2. A pdf file `report.pdf` describing the design of your classes – your choice of attributes, whether you provide any additional getters and setters, which universal methods you override, and any other design choices that you think are important to document. One of the members of the group must be clearly identified on the title page as the coordinator for this part of the project; the coordinator cannot be the same as in the first part (unless the group does not have enough elements). The source code for the developed classes should be included as appendix.

If you do not follow these rules, your project may be rejected and not graded. The report will be the basis for the evaluation.

Suggestions

It is recommended that you develop and test the four classes independently. You should test them by dedicated code, and you should also check that they follow the contract by integrating them with the other given classes.