INTRODUCTION TO PROGRAMMING
DM550, DM562, DM857, DS801 (Fall 2020)
Exam project: Introduction

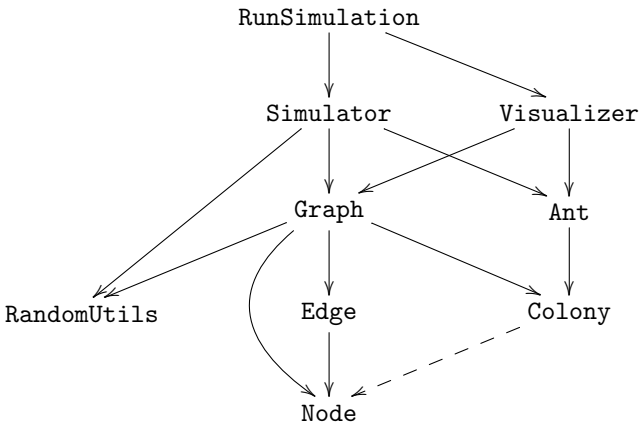
Biological analogies have been used for several years to find good approximate solutions for computationally hard problems. In this year's project, we look at one of these analogies: the ant colony problem.

Ants leave a trace of pheromones as they travel searching for food. These pheromones allow them both to retrace their steps back to the colony, and to find their way back to any interesting food sources they find. Pheromones degrade over time, so the paths taken by more ants are smellier, and more attractive for other ants to follow.

In this project, we abstract from the real scenario by considering that the ants move in a graph. In other words, there is a fixed, finite, set of *nodes* – places where an ant can be at any given time – and a set of connections between nodes – called *edges* – representing the possible moves that an ant can perform in one time unit. Nodes may also correspond to the location of ant colonies, or they may hold sugar reserves; edges have a level of pheromones, which increases when an ant passes by, and decreases with time. At any moment, ants decide where to go based on the pheromone levels of the edges departing from the node where they currently are at.

The program you will develop simulates the evolution of an ant colony. Time evolves in discrete steps; at each step, we compute the new position of each ant and update the sugar and pheromone levels in each node or edge at the graph. In order for the ants to be able to survive for a long time, there is also a small probability that a new sugar reserve will spontaneously appear at a node.

The class diagram for this implementation is given below. The solid arrows represent client/provider relationships (with the arrow pointing from the client to the provider). The dashed arrow is a different kind of relationship that will be explained later in the course.



The datatypes implemented in these classes are as follows.

- Instances of `Node` represent possible locations where ants may be.
- Instances of `Edge` correspond to existing paths between two `Node`s.
- Instances of `Colony` represent special `Node`s that also host an ant colony.
- Instances of `Graph` correspond to "maps" describing a set of possible locations (`Node`s) and legal connections between them (`Edge`s).
- Instances of `Ant` represent individual ants.
- Instances of `Simulator` correspond to on-going simulations, with a set of instances of `Ant`s placed in different `Node`s on a `Graph`.
- `RandomUtils` is a utility class containing static methods to generate random numbers according to predefined distributions.
- `Visualizer` is an auxiliary class that provides methods to visualize the current state of the simulation.
- `RunSimulation` is the executable top-level class, which is responsible for interacting with the user, setting up the simulation, and managing the main simulation loop by creating and storing instances of its client classes and invoking the appropriate methods on them.

Classes `Node`, `Edge`, `Colony`, `Ant` and `Graph` are motivated directly by applying a data-centred approach to the problem at hand. Class `RandomUtils` is a general-purpose utility class, which should be kept separate as it can be reused in other projects. Classes `RunSimulation`, `Simulator` and `Visualizer` could, in principle, be merged in a single class, and their division is motivated by two specific concerns. Separating `RunSimulation` and `Simulator` allows a cleaner separation between interfacing with the user and initializing all objects (done in `RunSimulation`) and implementing the progress of the simulation (done in `Simulator`). This also allows the work in these classes to be split between two different phases of the project, achieving a more balanced workload.[1] The methods provided in `Visualizer` would make perfect sense as methods of `Simulator`; however, implementing them requires working extensively with Java's graphical libraries, which is not in the scope of this project. Separating these two classes provides a modular way of dividing the corresponding programming tasks.[2]

The project is divided in three phases.

Phase 1. In this phase, you will develop the top-level class `RunSimulation`.[3] All the remaining classes will be provided in compiled form, together with their documentation. A significant component of the work in this phase is understanding the contract for all provider classes and how to work with them.

Phase 2. In this phase, you will develop the lower layer (classes `Node`, `Edge` and `Ant`), following the contract previously defined, as well as the remaining tasks regarding the simulation (class `Simulator`).

Phase 3. In this phase, you will develop the intermediate layer (classes `Colony` and `Graph`).

As discussed above, you will not need to implement classes `RandomUtils` and `Visualizer`.

In each phase you will receive more detailed information about the methods that you need to implement, as well as some implementation tips and examples.

[1] In an industrial scenario, a similar division could be used to allow for distributing the work between two different teams.

[2] Again, this would be an argument for splitting these two classes also in an industrial scenario, allowing them to be split among teams with different expertises.

[3] For students from DM562, this is the only phase of the project.