TÉCNICO LISBOA

Object Oriented Programming

2nd Semester 2017/2018

Project Report

Group:32

– Marco Montez. Nº 78508
– Tomás Cordovil, Nº79021.
– João Alves, Nº 78181.

Electrical and Computer Engineering Department

Contents

1 Project Goal

The goal of this project is to program a Java solution to the problem presented in the project statement using evolutive programming with stochastic mechanisms implemented with objects.

The problem consists on finding the best path, between a starting and end point, across a given n *by* m grid, with given edges, special cost zones and obstacles. By using the evolutive programming approach, the program spawns individuals at the starting point (initially), and lets them move, randomly, through the grid, given that they cannot move through obstacles. Each individual may, reproduce, move or die. Each individual has a certain comfort (based on its path and some pre-determined parameters) and can give birth to children: new individuals whose path is composed of 90% of their father's path at the time of their birth and 10% depending on it's comfort. If the number of individuals rises above a pre-determined value, an epidemic occurs, possibly killing all individuals except those with the 5 highest comfort values. The probability of survival of the rest of the individuals will be judged based on their comfort

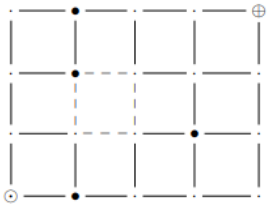

Figure 1.1: Example: 5x4 Grid

The simulation ends when no individual has any programmed events (reproduction, move or death) or the simulation reaches the final instant, τ. The best path is that of an individual that reached the final point (if any did it) and has the lowest path cost associated with him. If no individuals reached the final point, the best path is that of the Individual with the greater comfort associated with him, no matter if they're dead or not at the time of the final instant (τ).

2 Our Solution

According to the project statement we implemented various packages containing multiple classes and subclasses as described in the Javadoc file. We recommend reading that file as a complement to this report.

Below we will present some of the choices we made regarding Data Structures, Implemented Functionalities and some of the Object Oriented Programming principles that were applied.

2.1 Data Structures

During the project development we made some choices regarding the data structures used to store different types of data. We'll present the more relevant ones below.

2.1.1 Grid

After reading the XML data, our program stores the grid as an 2D-Array of objects of type *Point* (this objects stores within the nearby edges connected to that point as well as the *Point*'s corresponding x,y coordinates). This is possible as we know the number of points right a the start of the program the the XML file is read.

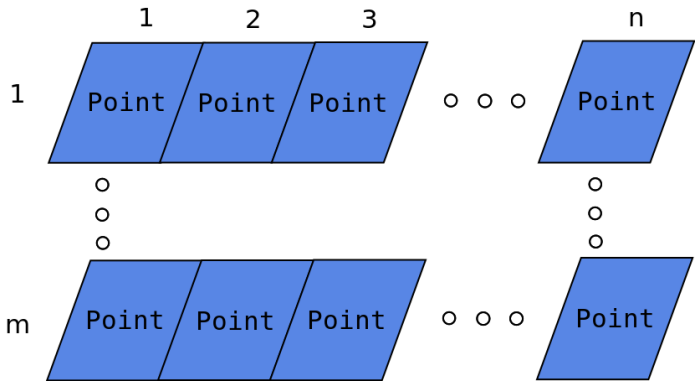

Figure 2.1: *Point* Array representation

Each *Point* contains an object named *NearbyEdges* that contains four objects of type *Edge* containing the *Points* it connects as well as it's cost. If that point is on the board extremity some of the *Edges* may be set to *null*. If a *Point* is an obstacle, the program sets it's nearby edges to *null* as well as all other edges contained in points in the array that are connected to that obstacle.

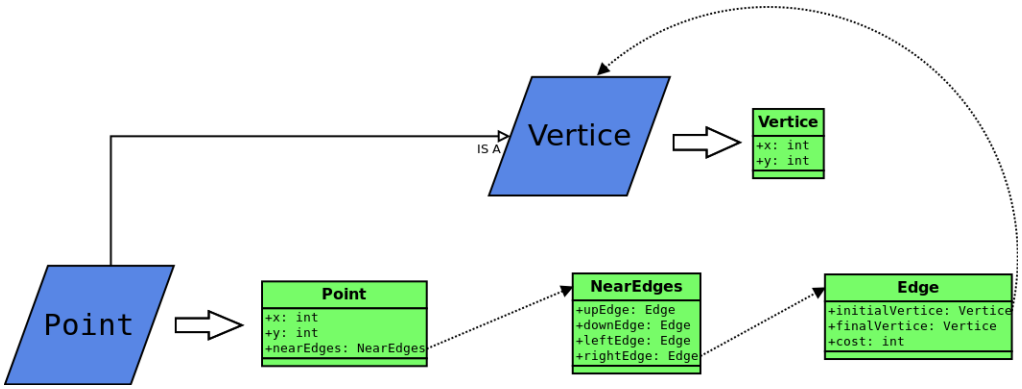

Figure 2.2: *Point* construct representation

Note: the group acknowledged the redundancy in declaring the x,y coordinates of *Point* twice (as an attribute of itself and an attribute of its superclass). This was implemented this way due complications using the *Point.Equals* method.

2.1.2 Individuals

The *Individuals* are stored as a *Linked List*. We chose this implementation because the number of *individuals* is not known *a priori* and a list ensures that expandability of the number of *Individuals*. Each individual is associated with the simulation, with a *Point* that contains its coordinates, with a path that is a *Stack* that contains all the points that *individual* has traveled by (by using a *Stack* we ensure that the last *point* on the *individual*'s path is always accessible), with its path cost, with its path length (hop count) and with its comfort.

2.1.3 PEC

The PEC is implemented using a *Priority Queue* with a comparator using the time of the event as the insertion criteria. This way we make sure that the next event is always at the front of the queue and we only

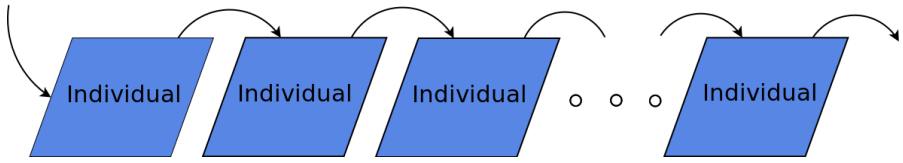

Figure 2.3: *Individuals* list representation

need to remove that (front of the queue) event when it is simulated. We chose this implementation due to the fact that the maximum number of events that can happen is known at the start of the simulation and, consequently, the structure that holds the events can be initialized with that size. Also, there's performance improvement in the access to the data, when comparing with other data structures that could be used to implement the PEC. This happens on account of the *Priority* aspect of this structure that allows us to insert, in order, a new *Event* onto the *Priority Queue*.

2.2 Implemented Functionalities

All the required functionalities are implemented in the project.

The PEC is correctly implemented as Priority Queue and performs as expected, each Event is inserted orderly based on its time may be on of four types of events we created (Move, Reproduction, Death and Observation). We implemented Observation as an Event although an Observation does not increment the Event counter (our iteration variable).

As said before, we implemented a Linked List to store the Individuals and the Epidemic mechanism works as expected, leaving alive the 5 Individuals with best comfort values and evaluating the probabilities (based on their comfort and a random number) of the rest.

The program usually finds the best path through the grid depending on the input parameters used.

Overall, the program implements all the functionalities required in the project statement.

2.3 OOP Principles

2.3.1 Encapsulation

We used the principle of *Encapsulation* on our project, when setting the visibilities of the different Objects, Attributes and Methods. We made sure that the project was closed to modifications by limiting the access to certain elements. For example, the visibility of the attribute *time* of the *AbsEvent* object is set to *package*, while the method *getTime()* is set to *public* so that the attribute can be read from outside of the package *Event*.

2.3.2 Inheritance

The principle of *Inheritance* was used in our project in different instances. For instance, *Point* inherits from *Vertice*, as it reuses all characteristics of *Vertice* and then some that are only necessary to the *Point* object. This can be seen in the UML model delivered annexed to this report. Another example of the application of this principle is the specialization of the different *Events* to the abstract class *AbsEvent*.

2.3.3 Abstraction

Abstraction is applied to our project, for example, in the *AbsEvent* abstract class that can receive one of four different events (*Reproduction*, *Move*, *Death*, *Observation*), inheriting the super class abstract method *simulateEvent()* and implementing it in different ways within each *Event*, therefore also using *Polymorphism*.

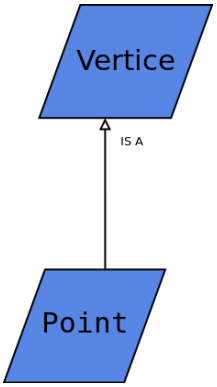

Figure 2.4: *Inheritance* principle representation

2.3.4 Polymorphism

As referenced above, we used *Polymorphism* in the interfaces we implemented. One example is the *IEvent* object that contains methods (*simulateEvent(), for example*) that is be used by different objects, in this case, the different *Events* we created.

3 Critical Evaluation

Although we implemented all the requested functionalities with success, there are some aspects of our code whom we are not entirely satisfied with. The group acknowledged the redundancy in declaring the x,y coordinates of *Point* twice (as an attribute of itself and an attribute of its super class, *Vertice*). This was implemented this way due complications using the *Point.Equals* method that requires attributes of the class *Point* as parameters and not of its super class.

Also, when implementing the *Observations*, we chose to implement them as *Events*, although they do not increment the counter of events (our *iteration* variable).

Another nuance in our code, is the fact of the last *Observation* is made at the instant *finalinst+1* to ensure correct final data output. This has no impact on the performance or correctness of the program execution.

We thought we could improve our code by implementing the *NearEdges* of the *Point* object as a linked list instead of the current implementation as it would save some memory space (on extremity *Points* we have *Edges* set to *null*) and we could also reduce the lines of code necessary to treat the data contained inside *NearEdges*.

4 Conclusion

In conclusion, this project was fit to test our abilities and comprehension of the principles of Object Oriented Programming, and educate ourselves regarding the different adversities that rise when tackling problems using this approach compared to other programming paradigms.

We learned a new approach to tackle path finding problems, using evolutive programming with stochastic mechanisms, in contrast to other algorithms we learned in past courses (Dijkstra's and some other BFS or DFS algorithms).

We also became familiarized with new tools we had never used, such as the *Eclipse IDE* for software development and the *Visual Paradigm* software for the UML model design.

In comparison with non-object oriented languages that we used in the past (like C language), we found planning and developing the software clearer and easier to organize since designing the UML model early allowed us to spend less time in architectural nuances when coding. Also, by using inheritance and polymorphism a good chunk of code was reused making its development easier.