PROGRAMMING SKILLS COURSEWORK

A PREDATOR-PREY SIMULATION

SARAH BEGGS

XIAO LI

COLUM ROZNIK

7 NOVEMBER 2014

THE UNIVERSITY OF EDINBURGH

INTRODUCTION

The purpose of this project was to create a program to allow a user to simulate hare and puma populations within a defined landscape where the respective populations were modelled by the following partial differential equations:

$$\frac{\delta H}{\delta t} = rHP + k\left(\frac{\delta^2 H}{\delta^2 x} + \frac{\delta^2 H}{\delta^2 y}\right)$$

$$\frac{\delta P}{\delta t} = bHP - mP + l\left(\frac{\delta^2 P}{\delta^2 x} + \frac{\delta^2 P}{\delta^2 y}\right)$$

H represents the hare population, P represents the puma population, and r, a, b, m, k, and l are all constants that can be altered by the user before the start of the simulation. The simulation modelled these continuous equations using a discrete approximation over a landscape partitioned into squares. We worked as a team to achieve this result and wrote our code using the Java language.

PLANNING

In our first meeting we briefly touched upon our previous programming experience and what each member of the group was best suited to contribute. We decided to take a more open approach to planning and Colum and Xiao immediately started writing some base code for the project to present at our second meeting. At the second meeting it was decided that the group would expand upon Colum's code for the rest of the project. Colum had created the Population, GridMap, and TestDriver classes at that point that would eventually be presented in the final code after modifications by all group members. To guide our efforts we drew up a list of tasks that the program must perform to meet the project requirements. This list guided the group's work and kept everyone on task.

GROUP ORGANIZATION AND TASK ALLOCATION

We met at least once a week to discuss and review where we were with the project and talk through any design or coding issues we had come across.

We chose to have a quite flexible and fluid allocation of work. We are on three completely separate degree programs and have different sets of courses and commitments so we decided we should try to work around this as much as possible. This meant that we knew that some weeks certain people would be able to do less work than others and we divided the work for that week accordingly but tried to make sure that that every person did a fairly equal share of the work over the course of the project.

For the final project, Sarah wrote the code for the program to read in and print data, as well as the classes to hold the hare and puma population characteristics and the unit tests. Colum wrote the main method, the code to hold information on the landscape and the hare and puma populations at a given point in time, as well as the algorithm to update the population for each time step. Xiao wrote the build file and contributed code improvements where needed and performed the final analysis of the program. Finally, all group members contributed to the final report.

DESIGN

In the basic design of our program, and in keeping with the principles of object-oriented programming, we organized the program by the basic data it holds and the functions that it performs. The program has classes to hold the characteristics of the puma and hare populations, a representation of the landscape, the number of animals across the map at any point in time, and methods to read and print data files.

Colum and Xiao both started the project by writing the base code independently before it was decided that the project would continue by integrating Xiao's code into Colum's initial code.

In the GridMap class program holds a two-dimensional array representing the landscape as 1's and 0's. The GridMap class has an isDry method to tell whether a given square is dry or not as well as the getDryNeighbors method which returns the number of dry neighbours adjacent to a given square. These two methods are both called upon elsewhere in the program including in the Population class.

The Population class has two double arrays to hold hare and puma population densities across the landscape. Of interest in the Population class is the algorithm in

the updatePop method, which uses a discrete approximation of the above differential equations to update the hare and puma populations for every time step of size Δt. The updatePop method takes in Puma and Hare objects as parameters, which hold information on the hare and puma populations, and makes use of several other methods such as the getAdjHarePops and getAdjPumaPops methods, which return the sum of the adjacent squares' populations. Because the landscape is represented discretely the simulated animals may only move in the four cardinal directions.

In our first draft of our code information about hares and pumas was integrated into the Population class. We found this rather undesirable in the long term as we felt that the methods in the Population class should be able to deal with any predator and prey system that followed the same structure of the equations. To this extent Sarah created the Animal superclass to hold the general properties of an animal including type (predator/prey), birth rate, diffusion rate, mortality rate, and predation rate. The superclass ensures that only an animal of type "predator" is assigned a predation rate. The use of a superclass rather than an interface was to allow explicit coding of the methods all Animal objects can have. The set methods for the various rates ensure that the user cannot enter a negative rate. This meant that the classes for Hare and Puma that defined the properties of these particular animals were quite bare and only included some specific constructors. A more complicated structure of an Animal superclass with "prey" and "predator" subclasses were considered, but we decided that this only made the project unnecessarily more complex since the Animal superclass was more than sufficient. The Animal superclass was designed for a predator-prey system, but it is general enough to model an omnivore animal that is both predator and prey. However, modelling an omnivore would require a different Population with a new updatePop algorithm, but the other classes would still function in keeping with an object-oriented program.

The PrintMethods class contains a number of static methods to print different information about a population to a file. The printingDenstityFile method prints the average densities at a given time for the predator in the 1st column and for the prey in the 2nd column to a file specified by the user. This allows the user to view the densities of the two animals side by side in the same graph and see how the two

populations interact. Average density was defined as being the number density in a square relative to the total animal population. The population densities must sum to 1 and failure to do so is a clear flag of a problem in the code.

The PrintMethods class also has a method to convert a double value into an RGB integer value. This method is completely general and normalizes a given number by the maximum possible value. So in a population grid n is the number of animals in a given square and if N is the total population, our normalized population is the number density in the square. This number density is multiplied by 245 and added to 10, to ensure any square with some population is at least slightly coloured. So a square with no population would be completely white with a value of (255, 255, 255) and a square with the entire population would be all black with a value of (0, 0, 0). We presented the hares as blue and the pumas as red so squares where they overlap will be purple.

There is also a method to print the density grid to a plain PPM file, which creates a single colour PPM file. Each animal is represented with a separate colour to make it easier for the user to watch the individual animals and changes. The magic number 'P3' is on the first line, the next line is blank, the next line is maximum RGB value, and the next line contains the dimensions of the population grid. The rest of the file is a matrix with the name number of rows as the population grid and three times the number of columns. As the RGB values are separated into three columns per one column of the grid each square is represented by one pixel.

The method to create the matrix is separate from the print method for the matrices. So we can print PPM files to look at the hares and pumas separately and we can also print the combined maps of their interactions. The only trick to the addition is to make sure that adding 255 is the same as adding 0 (white added to white is still white).

When printing out a PPM using the PrintMethods class, the user can refine what quantities they are plotting to try to get something that gives more visually useful results; currently this must be done manually in the code by recompiling. Ultimately RGB values are rather limited in what information they can capture so the full complexities and differences of the system will not be captured this way. The

PPM files can only visualize fairly large-scale changes and so for a big map the user will not be able to see visually if the population of a square has changed by much. To view fine changes in the system the user would want to use either the given density files or new ones for the quantities they are interested in to view the results as a graph. Printing of PPM files can give some information on the code but it is a crude tool that only gives limited information. The code was designed with a scientist in mind, someone who is more interested in the data from the density graphs rather than fine data from the PPM files. For more general users who wish to watch the changes more accurately the PPM could be defined to see more precise changes in the populations.

The ReadingMethods class defines two methods for the user to set the initial simulation conditions. If the user does not want to manually enter the initial conditions on the command line, he/she can feed an input text file into the program setting up the simulation. The readInitialValuesFromFile method only deals with setting up the initial conditions by setting proportions of hares and pumas across the landscape.

If the user does not specify any arguments then the user will be lead through a series of questions that allow her/him to change individual values one and at time. They can also choose where they want the initial populations of hares and pumas to be precisely by setting the population in individual squares.

Finally, the main method in the TestDriver class pulls all these other separate classes together and starts by calling the ReadingMethods class. The program allows the user to either feed in a file of initial population parameters or to enter them manually as a destination to print the data from the simulation. After creating Puma and Hare objects using information from the ReadingMethods class the simulation starts. Finally, the runtime for the simulation is printed out at the end of the program.

PROGRAMMING LANGUAGE

We all indicated a preference to work in Java given our previous experience working in that language. We found that Java was a more than versatile enough language to write a robust program to handle the simulation. It is interesting to note that the language was also flexible enough for our group members to work on the

project across several different operating systems and IDE's without significant issues. This is likely because Java is a high level language that does not require its users to interact a great deal with the underlying computer system.

Revision Control

We agreed to use a GitHub repository for our project, which is where we stored our work. Colum set up a public GitHub repository and limited access to collaborators only. GitHub proved to be an easy to use tool for our project and we did not encounter any major issues and were able to painlessly recover a file that one of our group members had accidentally deleted.

Build Tools

For this project, ANT was chosen as the build tool and a build XML file has been generated to run the simulation.

Just like a Makefile for the C language, ANT is an efficient tool widely used to build projects. It has a variety of functions, such as compiling classes of Java, defining tags and configuration files, and generating web packages to submit to Apache Tomcat. It uses XML to describe targets and set properties for each task.

The biggest challenge with ANT is that the syntax can be quite cumbersome in the build file. However, since the use of ANT is so widespread an issue of syntax can be resolved by consulting internet reference sources.

For this project, some targets are used to implement different functions such as init, compile, run, and clean. The targets of init and clean were implemented to make and remove build directories, compile and compileTests to compile source and test code, run and runTests to execute classes, and jar and jarTests to create jar files. The use of MacroDef not only reduces duplication of effort, but also improves the efficiency and the reusability of code, and guarantees ease of maintenance. The test results can be written to XML and HTML, which allows the user to view them through a web browser.

Therefore, the build file can be used as “write once run many” tool. It manages code more reasonably and allows the structure of the project to be easily extended.

TESTING FRAMEWORK

We made sure that we were all involved in testing the code. Unit tests were created for the various classes we created to make sure that we were getting correct values for certain methods and that exceptions were being thrown as we wished. We added unit tests for incorrect user input, for example testing that for negative population characteristics the program throws an IllegalArgumentException and testing on a small grid that simple methods such as getDryNeighbors work correctly. This meant when we had to debug larger methods that called these simple methods, we were sure that these component methods worked so we knew that the problem had to be elsewhere. Thus using unit tests from early on ensured we were not wasting time unnecessarily checking extra code. We added unit tests when we encountered simple bugs so these could be tested for when the code was changed to make sure these were not reintroduced by a modification.

Unit tests were not appropriate for all methods; we could only test some methods by running the entire code and viewing the outputs. We also knew that we could not imagine all the ways a user could misuse our code, however, we tried to think of as many as possible. We also experimented to see if errors were created by unrealistic initial conditions or by a problem in the code.

We also tested a variety of initial scenarios to see if the program handled them as we expected; this topic is expanded upon below in the performance and analysis section.

DEBUGGING

We initially encountered some bugs in the GridMap class that were mostly due to confusion involving the notation. Colum initially wrote the GridMap class and named the fields for the columns and rows as ny and nx to conform to the notation in

the assignment of N_y and N_x. Since the names of these fields are not very intuitive Xiao confused the two when she expanded upon the code in one of the constructors; Colum later identified this bug and corrected the situation by introducing new field names.

We later encountered bugs in the Population, PrintMethods, and ReadingMethods classes involving the updatePop method, which updates the populations, the code for reading in files, and printing to files. We all took part in searching the code for problem areas and identifying bugs by manually running the code and examining the output and using IDE debugger tools and techniques such as breakpoints and checking the call stack.

While IDE debuggers are very useful sometimes more rudimentary methods such as adding print statements were more helpful and efficient.

Performance Tests and Analysis

To optimize our project the first step in profiling was to measure speed and efficiency of the program. Timing by hand was the method we chose to reduce impact on overall code performance. We noticed that the simplest method to speed up the program would be to reduce the number of output PPM files since printing PPM files

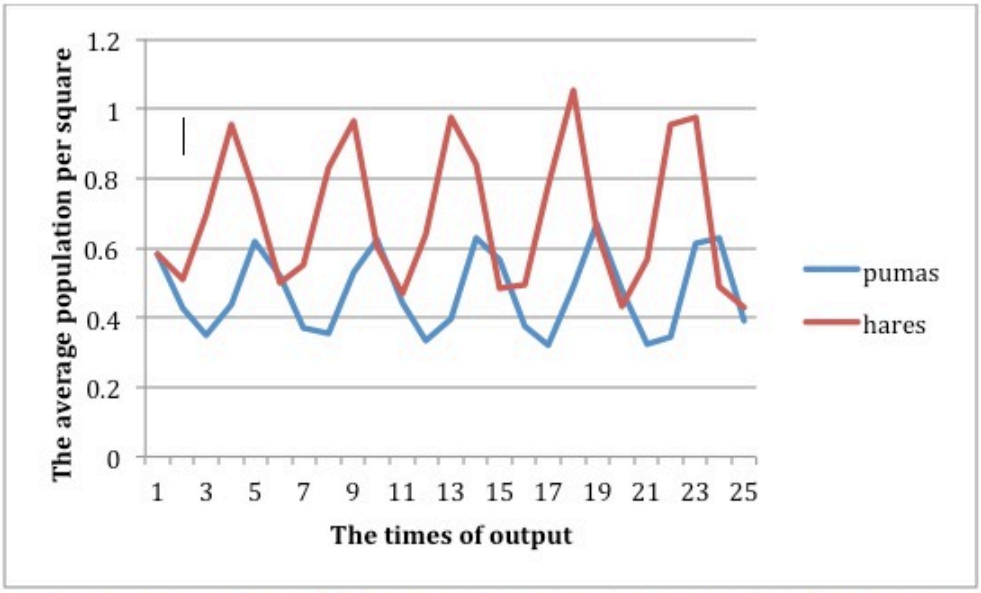

Figure 1. The average population per square of pumas (0 to 5) and hares (0 to 5) at different times in a 2000 x 2000 landscape.

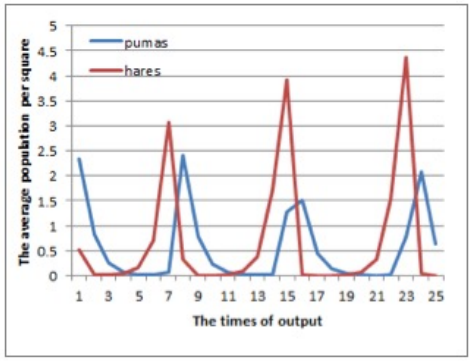

Figure 2. The average population per square of pumas (0 to 20) and hares (0 to 5) at different times in a 2000 x 2000 grid landscape.

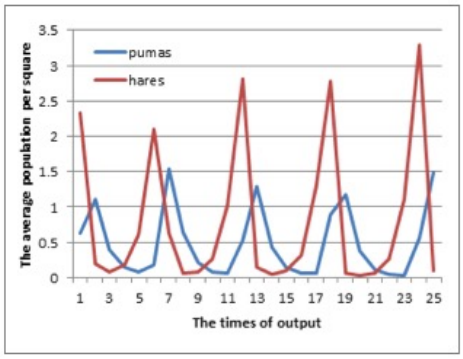

Figure 3. The average population per square of pumas (0 to 5) and hares (0 to 20) at different times in a 2000 x 2000 grid landscape.

takes such a long time. In addition, through adding some timing calls, the results showed that the updating part cost the most in CPU time, which is exactly what we expected. To improve performance, we could change the structure of the updatePop algorithm. For example, the program does not need to examine the water portion of the landscape in its calculations, which increases the program's runtime unnecessarily. Therefore, the project only examines the land portion of the landscape in the algorithm. The experiment was conducted with varying sizes of landscapes, which indicated that larger sizes increase CPU time.

In regards to the compiler, JVM of IBM can conduct mathematical computation very quickly. In addition, JVM of BEA has the best performance in handling a large number of threads and network sockets. However, we chose to use Sun's JVM because of its ability to work with data intensive programs. Although Java does not have as many flags to improve compiling performance compared with the C language, VM supports HotSpot compilation, which introduces the cache mechanism to store binary codes temporarily generated by Java codes with high frequency of being executed. The garbage collection is embedded in the virtual machine automatically, which clears unused memory.

From the experiments we conducted, we can draw several conclusions. Firstly we see a relationship between the two populations that one would expect to see in the wild; the hare and puma populations follow a similar pattern and are clearly interdependent.

To a certain extent, an increase in the number of prey will cause an increase in the number of predators, but the growth in the puma population lags the growth in hares. The reason for this, we suspect, is that an increased number of predators directly leads to a decreased number of prey.

Secondly, we experimented with large differences between the two initial populations. For example, the density of pumas takes on random values from 0 to 20 and hares from 0 to 5 (figure 2), and vice-versa (figure 3) where density is the population per square over the total number of squares. From the results of the experiment, we find that the initial population differences between predators and prey does not influence the overall trend of interdependence between pumas and hares. However, from figures 2 and 3, there are differences from figure 1. When there are more pumas the number of prey will drop dramatically, and meanwhile pumas will also decrease during this period of time, while in the opposite case, the large number of prey will lead to an increased number of predators.

Thirdly, when setting one population to zero we see different responses for prey and predators. When there are no pumas the number of hares grows exponentially in the absence of a predator (figure 4) and in the absence of prey pumas disappear from the landscape (figure 5). To make this a more realistic model, further work could rework the differential equations to add in a hare mortality rate and other variables to reflect the true complexity of a predator-prey system.

Fourthly, the GIF file was created from many PPM output files to represent the results. It indicates that no matter how uneven the initial densities of pumas and hares are, they will shortly distribute evenly across the landscape and their population numbers will follow a cycle.

Finally, an extreme scenario with landscape where none of the dry squares were adjacent to other dry squares was considered. However, because the landscape is discretely approximated this situation results in non-real birth death.

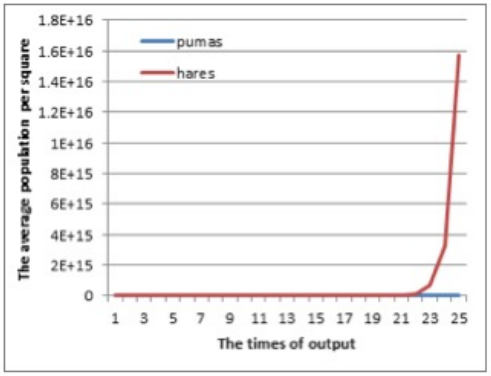

Figure 4. The average population per square of pumas (0) and hares (0 to 5) at different times in a 2000 x 2000 grid landscape.

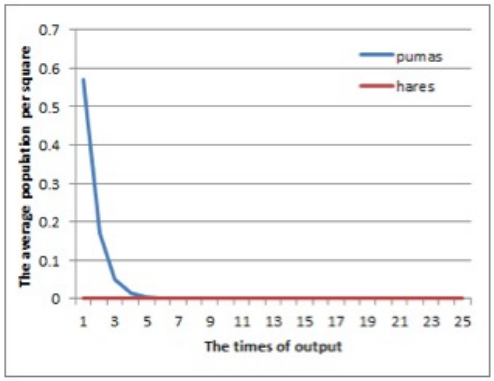

Figure 5. The average population per square of pumas (0 to 5) and hares (0) at different times in a 2000 x 2000 grid landscape.

CONCLUSION

This program reads data from DAT files and produces PPM files with different densities of the animals. When the user runs this program, they can use a default input file that includes default values by themselves.

In this project, git and GitHub were used to merge our code appropriately for revision control. ANT was chosen as the build tool to deploy our code automatically. Test units were used to test each class and ensured correctness. We debugged the program using IDE debuggers when we finished every function. Timing by hand was used to profile the program and experiments were conducted with different inputs to get results for our analysis.

FUTURE WORK

The speed and efficiency of the program would be vastly improved with the use of Java Threads.

We have tried several kinds of input methods for users to choose, and in the future they are expected to be more suitable and intelligent for users.

The output can be modified to generate more meaningful PPM and GIF files for users interested in viewing the visual impact across the landscape, which can be more explicit not only for users but also for our data analysis and testing.

In addition, we would have liked to further refine the PrintMethods class to provide a more nuanced colouring of the population PPM files to reflect better the density levels of hares and pumas. This tends to work with test data, but the algorithm needs to be adjusted to work real data.