|epcc|

Pumas And Hares Documentation

Group

November 4, 2018

Contents

1 Introduction **3**

2 Environment **3**
2.1 Tools . . . 3
2.1.1 Infrastructure . . . 3
2.1.2 Build Toolchain . . . 3
2.1.3 Development Process . . . 3
2.1.4 Included Libraries . . . 3
2.1.5 Unit Testing . . . 3
2.2 Build and Run . . . 3

3 Key Design Decisions **4**
3.1 Requirements . . . 4
3.2 Input . . . 4
3.2.1 Storage of Simulation Parameters . . . 4
3.3 Landscape . . . 5
3.4 Simulation . . . 5
3.5 Output . . . 5
3.6 Utility Classes . . . 5
3.6.1 Landscape Generator . . . 6
3.6.2 Configuration Generator . . . 6

4 Tests **7**
4.1 Unit Tests . . . 7
4.2 Scripts . . . 7

5 Tasks Arrangements **8**

1 Introduction

In this assignment, we implemented a sequential version of Pumas and Hares simulation using the model enunciated in the coursework specification.

C++ was used as the core programming language for the program (C++11 compliant). Auxiliary scripts using Python (version 3.5) and BASH (version 4.4) were used for automated and repeated testing of the core programs produced.

2 Environment

2.1 Tools

2.1.1 Infrastructure

The EPCC Cirrus front-end was used as primary development and debugging platform (CentOS 7.3, Linux 3.10, Glibc 2.17). The latest submitted release of the program should build, compile, and test correctly on Cirrus (as of 4/10/2018).

2.1.2 Build Toolchain

The GNU toolchain was used for building and debugging purposes, including GNU make (version 3.82), for automating builds and tests, the GNU Compiler Collection (version 4.8.5), for compilation, and the GNU debugger (gdb).

2.1.3 Development Process

Git (version 1.8.3.1) was used as main version control system, hosted in a private Github repository. Travis CI was used as a continuous integration framework, to incrementally and atomically test-build commits.

Developer documentation can be produced by using Doxygen on the project, as a result source and header files are appropriately commented.

2.1.4 Included Libraries

Three main libraries are included in the project:

1. CppUnit, used as a unit testing framework [1].
2. "JSON for Moden C++", used as a JSON parsing library [2].
3. "Args", used as a Boost-like commandline argument parsing library [3].

2.1.5 Unit Testing

As cited, CppUnit was used as the main unit testing framework.

2.2 Build and Run

Refer to the ReadMe for detailed build and run instructions

3 Key Design Decisions

3.1 Requirements

In order to run our program, it was established that several pieces of data are required. Along with the landscape file, we established that the use of a configuration file, for each simulation parameter, would be a useful and suitable addition, as will be further discussed.

3.2 Input

Complying with the coursework specification, the program takes a landscape file as input (-i option), composed of a header with the dimensions, followed by a grid of 0s and 1s, indicating water or land presence, from which to run the simulation.

Several Landscape input checks were decided upon. Firstly, the body of the landscape is checked, so that it complies with the dimensions specified in the header. The landscape file is also checked, so that only 0s or 1s compose the body. If either of these checks are not met, program execution stops.

Secondly, when running the program, a prefix name for the files with the average populations must be specified (-p option) and a JSON file with the configuration parameters must be passed (-c option). This file contains a simple dictionnary of each parameter described in the coursework specification, needed to run the simulation (a,b,d,k,l,m,n,r). These parameters are checked, so that they are positive. If a negative value is parsed, an exception is thrown. To additionally shield against negative values, the ConfigurationGenerator (which will be further discussed) utility binary only allows positive values

The above checks and inputs to respective data structures, are performed by the LandscapeParser class, and the ConfigurationParser class. The LandscapeParser class using the Landscape class as main data structure to store the state of the simulation between time steps. Limits are defined on the maximum dimensions of the landscape, being 2000 by 2000 squares, and the maximum time step T, set at 500.

3.2.1 Storage of Simulation Parameters

As opposed to passing the simulation parameters via commandline arguments, storage in a configuration file on disk was preferred, as this would improve user experience, allowing for multiple simulations with the same parameters can be run on different landscapes, without needing to manually pass each parameter. In addition, configuration files can be named, which would help additionally in managing multiple simulation runs. Finally, less commandline arguments would potentially lead to less user-errors when running the program. As opposed to using XML or BSON, we opted for JSON due to its balance between fast parsing speed and human-readable properties.

3.3 Landscape

The Landscape class provides the state of the Landscape for the rest of the program. It is instantiated and populated by the LandscapeParser class, and used as the state from which to perform the simulation steps. The landscape parser uses the landscape file header, to allocate a static array of landscape squares and populates it with land or water, if the checks previously described pass. This stores the simulation parameters, as member variables, and the grid representation of the landscape (the static array), composed of individual squares, instances of the LandscapeSquare class. The LandscapeSquare class is the representation for each individual square in the grid, and contains information on the square it describes, such as:

1. If the square has land or water.
2. If the square has land: the population numbers of pumas and hares in the square.

The selection of data structures to perform this was not trivial. Using a defined LandscapeSquare class, along with minimising the use of dynamically allocated data structures was done in order to reduce memory usage.

3.4 Simulation

The main core of the simulation resides in the LandscapeSimulation class. This class takes an instance of the Landscape class, previously populated and configured with the LandscapeParser and ConfigurationParser, updating in-place the state at each repetition, using the enunciated equation in the coursework specification to use when calculating new Puma and Hare numbers between simulation time steps.

This design allows for separation between the data, the state, and the logic to be performed.

Several checks are performed from within the LandscapeSimulation.Run() function, the core implementation of the equations provided by the coursework specification. It is defined that no negative population values are allowed in the state, when a negative population value is obtained, it is defaulted to 0 instead. We acknowledge that this would alter the decline in population of the opposite species, however we established that negative numbers should not be used as they do not reflect a real, plausible state the populations could have.

3.5 Output

The outputs of the simulation include images of the landscape every T time-steps in ppm format. This T value is provided in the JSON file for the configuration of the landscape and is restricted to be less than the value 500. The output images are represented using 4 colors. Each cell has a specific color according to the population of animals in the cell. If pumas are more then the cell is red whereas if hares are more the cell is green. If the cell is water then the color is blue. In case where the concentration of pumas is equal to that of hares the color is black. In addition the algorithm outputs an extra text file with average densities of pumas and hares across the area of the whole landscape.

3.6 Utility Classes

In addition to the main simulation program, generators for valid configuration and landscape files are provided. Once compiled, these utility binaries help to demonstrate working examples, and are of extensive use when performing tests using the BASH commandline.

3.6.1 Landscape Generator

This class takes as inputs the dimensions of the landscape as height and width correspondingly from the command line. The dimensions of the landscape when given in command line should be integer non-zero values in the range (1-2000) for each of the dimension. A user-defined value between 0 and 1 is provided for the land/water density. The algorithm calculates the total land area (totalSpace) in the landscape from the dimensions and then calculates the number of land (landPoints) that should be added (integer – rounding down) in order to get the analogous concentration. At first, the land is thrown into the whole area "pseudo-randomly" in blocks of 3 by 3 until the total number of land in the area reaches the value (landPoints). The 3 by 3 blocks might be overlapped during the random creation but this is taken into account from the algorithm. In the case that a land point is created without neighbors then there is a part of the algorithm that checks the whole area, finds it and adds next to it one more land space. Later on, it finds one land point with more than 2 neighbors and makes one of them water in order to "bring back" the correct land/water balance from the initial creation. After the final checks described above the landscape is then written in a file . This file contains in the first line the height and width of the landscape (dimensions) and then the landscape.

3.6.2 Configuration Generator

The configuration generator consists of a utility class, that would help the user generate a JSON file with all the required user-defined parameters, in addition to check if the user-provided values for all parameters are legal. The ConfigurationGenerator has a standard getter-setter design, with the constructor taking commandline arguments (argument count and vector), setting all parameter values. Finally, a method is called to write these values to a JSON file using a stream, named by the user. The generator makes basic input checking, including type and sign of the values provided.

4 Tests

4.1 Unit Tests

Unit tests were implemented for a majority of the classes produced. The main rationale behind them is to ensure that Setter functions correctly give exceptions, when illegal values are provided, and that reads and writes of files result in the expected exceptions, or are performed correctly, when all necessary requirements are provided.

The main mechanism for our unit tests is by the use of assertions and exception catching. When testing a setter with an illegal value, a true CppUnit assertion is made, at the same time that exceptions are catched. If no exception is catched, a false CppUnit assertion is made, and the test fails.

Unit tests can be run from the commandline with GNU Make, please refer to the ReadMe for further instructions.

4.2 Scripts

A set of scripts were written in order to further automate testing and validating of our simulation using various size for the landscape and diffrent configurations. Using the previous mentioned scripts, we generate 9 different configurations and 16 landscapes(generateConfigurations.sh,generateLandscapes.sh) and after that we run our simulation for 144 times which are all the possible combinations.(runSimulations.sh). Finally, we validated our outputs using validateSimulations.sh and validation.py. Please refer to section Run multiple simulations (optional) at ReadMe.md for more information.

The validation script (validation.py) is used to perform several checks on simulation landscape outputs, directly on the images produced by our simulation program. It checks that the dimensions of the image match the dimensions of the landscape, it checks that the averages obtained from the simulation are positive numbers, and finally, that the pixel totals for water and land match (the image produced must have the same water pixels, as the ones given in the landscape input file).

5 Tasks Arrangements

We divided our project into the following tasks:

1. Design of the project
2. Implementation of landscape parser
3. Implementation of configuration parser
4. Implementation of landscape generator
5. Implementation of configuration generator
6. Implementation of simulation
7. Implementation of output
8. Implementation of validation
9. Implementation of unit tests
10. Implementation of scripts
11. Documentation

Tasks Arrangements	
Kavroulakis Alexandros	1,2,6,7,8,9,10,11
Andreas Hadjigeorgiou	1,2,4,11
Angel Millan Campos	1,3,5,9,11

References

[1] *CppUnit* https://freedesktop.org/wiki/Software/cppunit/

[2] *JSON for Modern C++*. https://github.com/nlohmann/json

[3] *Args*. https://github.com/Taywee/args