|epcc|

Software Development Planning and Risks

B098688 - s1671778

February 7, 2017

Contents

1 Introduction

This report will provide a series of guidelines on how to proceed with the creation of a web application. The web application's code has been initiated by an unknown party, that is written in `python` and uses the web framework `flask`. Said code must be enhanced such that it fulfills the base requirements for the project. The skeleton of the web application has a few elements already built in. It is, however, very rough and needs much work to become a feasible end product.

The first part of the report will briefly outline what is the intended project, and the background needed to understand what it is attempting to achieve. While this is being discussed, some of the important factors to keep in mind are highlighted. After this is done, several comments regarding issues in the provided code are made. Afterwards, solutions for most of these are proposed, and eventually presented in a *time/effort* estimation plan. Finally, once the project has been understood, and its purpose is clear, risk analysis and management strategy are posed.

2 Project Specifications

The goal is to further develop a web based application that allows the user to generate a squad for a tabletop game - with certain constraints. The application must allow users to create and edit squads with an unchangeable name, no more than 10 members, and no fewer than 1. The user begins with a Captain, that can become better with time, and has a limited amount of money to hire other regular team members, that cannot improve with time, and a special member, the Ensign, who, like the Captain, can evolve in time.

The idea of the web application is to check that all constraints are fulfilled for a user's squad, and that if one or more are violated, a warning message is displayed such that the user is able to modify their decision. The base stats for each team member are shown in the Annex.

2.1 New Squad

A new squad begins with **one** Captain and **500** credits. For a new squad to be created, the following conditions must be met:

1. Squad Name.
2. Positive number of credits.
3. Captain:
   (a) **ONE** per squad.
   (b) Starts with **one** EXPERIENCE.

(c) Must be assigned **one** SPECIALISM (Engineering, Psychology, Marksman, Tactics, Melee, Defence).

(d) **One** Associated Skill must be assigned to the specified SPECIALISM:

i. Engineering: [Repair, Sabotage, Augment].

ii. Psychology: [Bolster, Terror, Counter].

iii. Marksman: [Aim, Pierce, Reload].

iv. Tactics: [Squad, Ambush, Surround].

v. Melee: [Block, Riposte, Dual].

vi. Defence: [Shield, Sacrifice, Resolute].

(e) Must be given **one** WEAPONS/EQUIPMENT:

i. Blaster: **5** credits.

ii. Needle Gun: **12** credits.

iii. Blade: **3** credits.

iv. Cannon: **15** credits.

v. Whip: **5** credits.

4. Ensign:

(a) Price of hiring: **250** credits.

(b) Maximum **ONE** (squads can be created without an Ensign).

(c) Must be assigned **one** SPECIALISM.

(d) Must be assigned **one** Associated Skill.

(e) Must be given **one** WEAPONS/EQUIPMENT.

5. The total number of members is **10** (including the Captain and Ensign).

6. Prices of hiring are:

(a) Augment Gorilla: **20** credits.

(b) Lackey: **20** credits.

(c) Security: **80** credits.

(d) Engineer: **60** credits.

(e) Medic: **50** credits.

(f) Commando: **100** credits.

(g) Combat Droid: **150** credits.

In order for a new squad to be generated,

3 Identified Problems

Now that the constraints of the teams have been covered, it is possible to go through the code to check missing or wrong information.

This report explains a series of performance tests that were made for a Predator-Prey model written in Python 3. Said model attempts to describe the changes in population densities of hares and pumas over a given landscape. The model receives the following: a landscape where both animals live together that can contain land and water, the population density of each animal, a predation rate (pumas that eat hares), birth rates for each animal, mortality rate for pumas (hares do not die of natural causes), and diffusion rates for each animal.

The Predator-Prey model that was analyzed in this report has written by *Callum Black, Andrés Cathey, and Eskil Joergenssen* for the HPC course **Programming Skills**. This program exploits the resources of the python package `numpy`. Doing so it is able to analyze a 100% land-filled 2000x2000 landscape in less than 7 minutes.

The input file that is given to the program has 0s and 1s that represent water and land respectively as shown in fig. 1 (or 2) for a 30 by 30 file. When the input file is only water then neither hares nor pumas can live in it, and barely no computations should be done. This is the motivation for the first performance test that is done for this program, i.e. how is the program's run time affected by the land-to-water percentage for a given input landscape size, expecting the run time to increase linearly as more land is added to a fixed size input landscape.

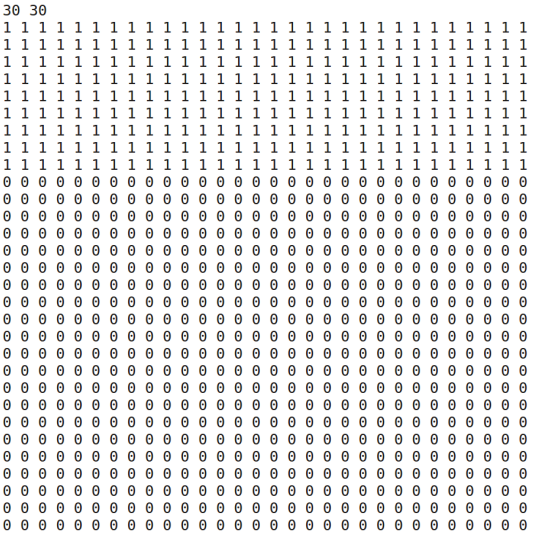


Figure 1: An input file with a block of land (30% land).

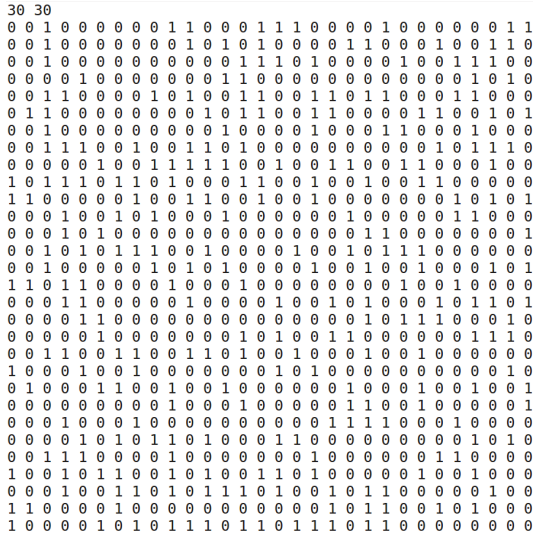


Figure 2: An input file with randomly filled land (30% land).

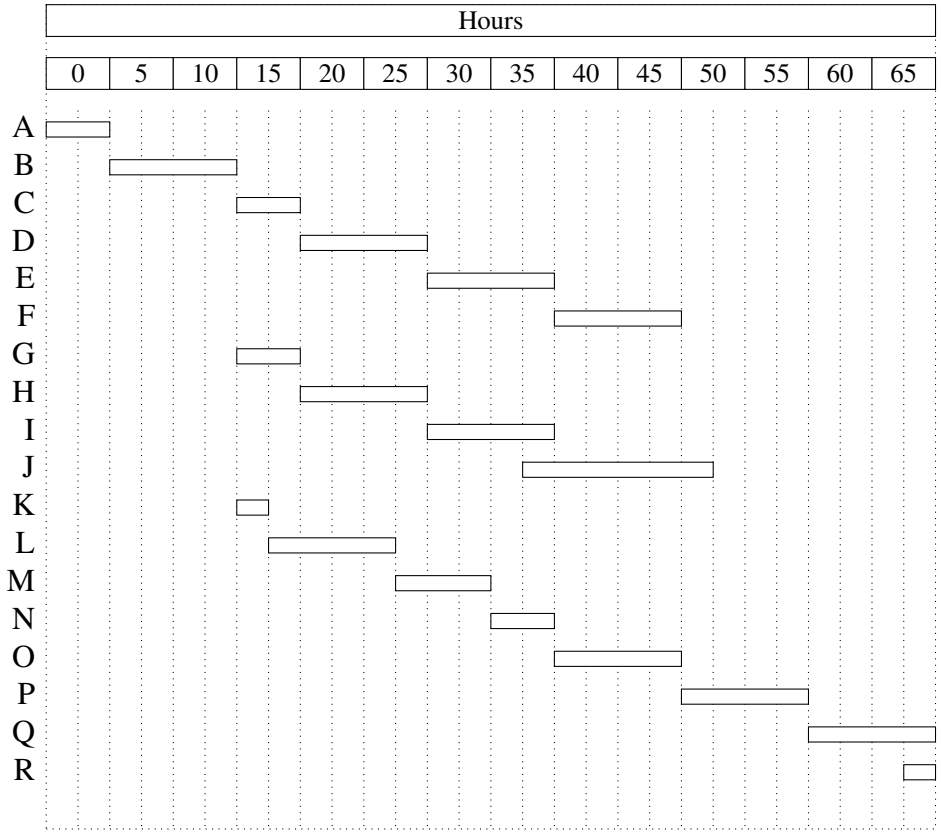
The "second" performance test is a slight variation of the first test. It is done by taking two different ways of filling the landscape with land. The first way to do this is by

creating a block of land and a block of water. A different way to fill the landscape is to do it in a random manner (without changing the percentage of land in the input file) as shown in fig. 2. This performance test is related to the diffusion rates of both animals (their movement within the landscape) - land squares in randomized files should, on average, have less land neighbors than land squares in 'block' filled files.

The third test that was done relates to the size of the input file. By changing the input landscape area and calculating the run time of the program we expect to see a linear growth - failure to observe this would mean that a significant optimization in the program. These were done first with a static land percentage, and afterwards with varying land-to-water percentages.

Finally, the program was profiled using the python native `cProfile`. This was done for input files of 100x100 and 1000x1000 and, in both cases, for 30% and 60% land-to-water ratios. This method was probably the most helpful in obtaining information on each function used. It helped determine where any optimization attempts should be directed at.

4 Project Planning



One of the important restrictions of the model is that neither hares nor pumas can live in sections that represent water. Therefore it is straightforward to expect that if a landscape

is entirely covered in water it should run faster than one completely covered in land, since no computations related to the population densities should be required. This is the motivation behind the first performance test.

For an input file of fixed size the percentage of land to water is varied from 0% to 100%. Using a bash script, the time it takes to run the program with different land-water proportions in a 100 by 100 landscape is plotted and fitted to a function as shown in fig. 3.

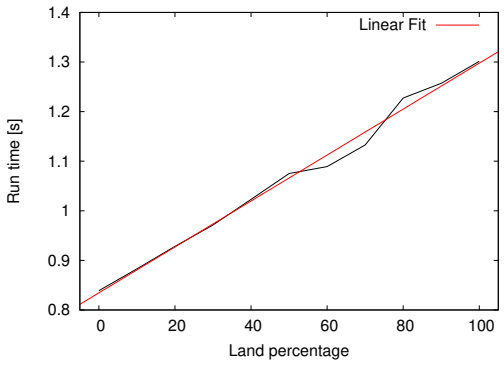


Figure 3: A plot of the run time depending on the percentage of land in the 'block' input file.

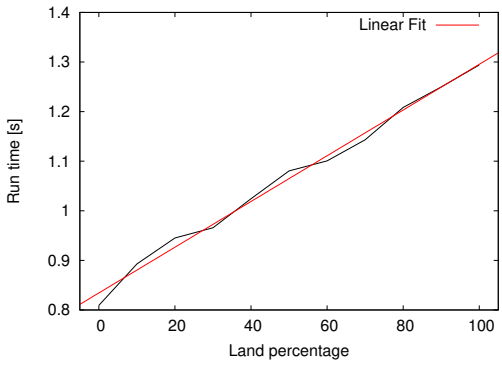


Figure 4: A plot of the run time depending on the percentage of land in the random input file.

The output confirms the intuition that the program's run time should increase as the percentage of land in the input landscape file is increased. Furthermore, it shows that the aforementioned growth has a direct proportionality to the increase in land-to-water ratio, following eqn. 1

$$RunTime(land\%) \;=\; 0.83451 \;+\; 0.00463\, land\% \tag{1}$$

4.1 Randomly Generated Landscapes

The previous performance test was done with "block" input landscape files, i.e. similar to fig. 1. The question of whether this would change if the input landscape files used were of the likes of fig. 2 is also briefly studied. By doing the exact same analysis as above with randomized land filled files gives fig. 4, with a fit defined by eqn. 2.

$$RunTime'(land\%) \;=\; 0.83434 \;+\; 0.00461\, land\% \tag{2}$$

Since the slope of eqns. 1 and 2 define how the computation time escalates as the land-to-water percentage increases, a simple comparison between the slopes of each equation provides an idea as to whether there is any significant difference between both methods. The slope of eqn. 2 differs by less than 1% from the one of eqn. 1. From this it is possible

to conclude that it is irrelevant whether the input landscape file was randomly generated or generated in a particular order, and the factor that needs to be considered is the ratio of land-to-water.

5 Input Landscape File Size

The most important aspect of the run time of the program is the size of the input landscape file. For the previous performance tests the size of the input landscape file was 100x100, which corresponds to a landscape area of 10000. It is of interest to calculate how does the run time behave when we linearly increase the landscape area. To do so a bash script is implemented that calculates the run time of the program for different landscape areas and plots these to check the nature of the growth.

Figure 5 shows a linear relation between the run time and the input landscape's area. Since the input landscapes used are squares, then a quadratic relationship between the run time and the input landscape's squares per side is expected, and, indeed, observed. This is shown in fig. 6. Both figures show the run time for landscape inputs of land-to-water percentage of 30%.

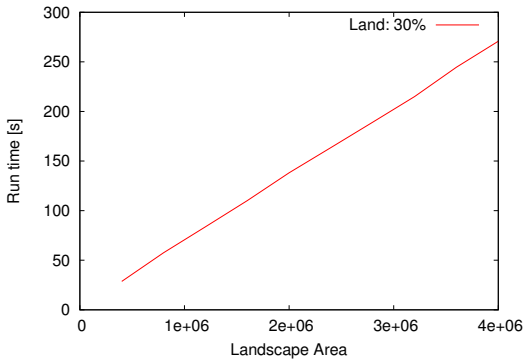


Figure 5: Run time vs. area with 30% land-to-water.

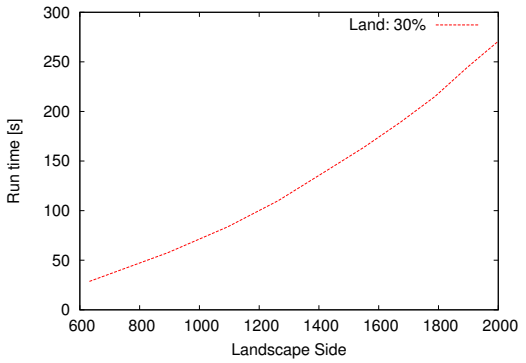


Figure 6: Run time vs. area with 30% land-to-water.

To check whether these relations hold for different land-to-water percentages a sweep for this is also generated. This is shown in figs. 7 and 8. Since the same relation appears for several other land-to-water percentages, it is possible to conclude that despite increasing the land-to-water percentage in the input file does increase the run time, it does so in a linear, and not exponential, fashion. In other words, the run time is directly proportional to the landscape area regardless of the land-to-water ratio.

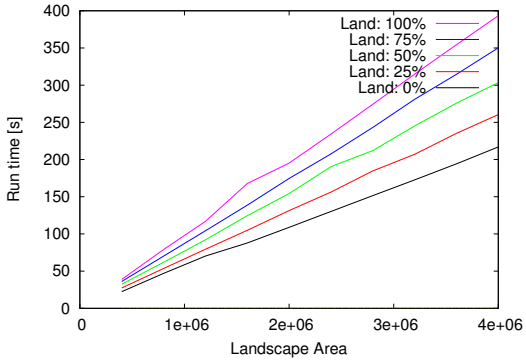


Figure 7: Run time vs. area with different land-to-water percentages.

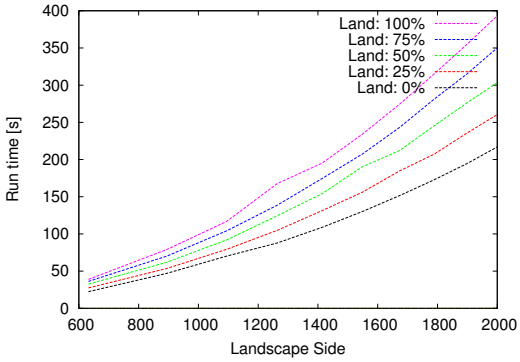


Figure 8: Run time vs. area with different land-to-water percentages.

6 Code Profiling

In order to actually know which functions make up for most of the time in the run time for the program, the python native profiler "`cProfile`" was used. First, two input files of the same size (100x100) one with 30% and the other 60% land-to-water ratios were profiled. The output shows how much time each function took and how many times they were called. These results are summarized in the pie-charts of figs. 9 and 10.

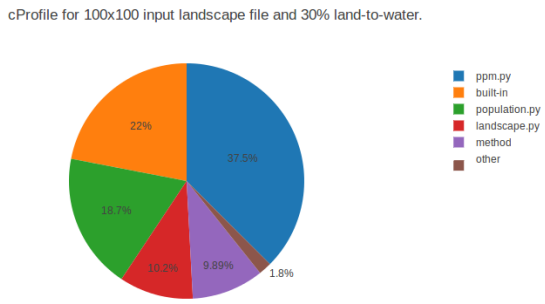


Figure 9: Pie chart of run time components for 30% land-to-water ratio in 100x100 landscape.

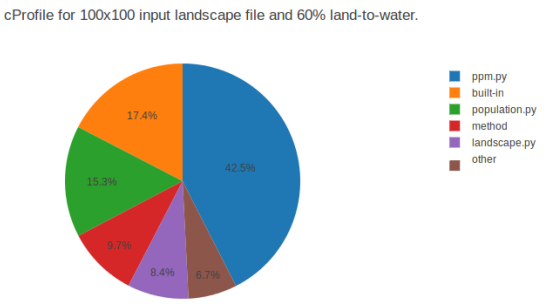


Figure 10: Pie chart of run time components for 60% land-to-water ratio in 100x100 landscape.

These images show that functions inside the file `ppm.py` are the most time consuming for both the 30% and 60% cases. The "`built-in`" and "`method`" are instances where `python` calls for certain functions, modules, or functions within them. Given that the contributions from python calling modules and such will become less relevant when larger files are used, we use the "`cProfile`" method once more. Obviously, this time the input landscape file will have a larger size - 1000x1000. This will help find out what components of the actual program are taking more time.

In this occasion, the landscape input files of 1000x1000 are also studied with land-to-water percentages of 30 and 60 are used.The corresponding pie charts are generated in

the same way as before (again with a bash script). These are shown figs. 11 and 12.

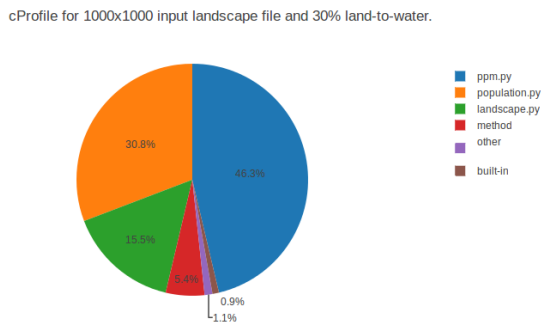


Figure 11: Pie chart of run time components for 30% land-to-water ratio in 1000x1000 landscape.

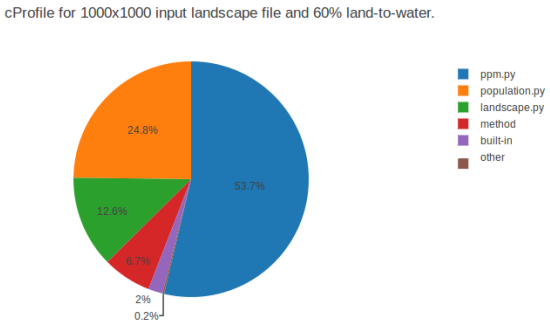


Figure 12: Pie chart of run time components for 60% land-to-water ratio in 1000x1000 landscape.

As expected, the pie charts for the larger input file (11 and 12) have a significantly smaller contribution from the "`built-in`" and "`method`" instances. Additionally, they show that the functions of `ppm.py` are the most time consuming regardless of the land-to-water percentage and the input file size.

These figures provide a more complete picture of what can be done to optimize the program under study. The first thing that is evident is that `ppm.py` is using most of the run time in all four cases. This file holds the functions that are used to generate the `.ppm` files. Hence this is where it would be most beneficial to attempt some type of optimization. Additionally, the second set of pie charts makes it clear that the files `population.py` and `landscape.py` are also good candidates for possible optimization.

7 Conclusions

After performing a series of performance and profiling tests on the program written by *Callum Black, Andrés Cathey, and Eskil Joergenssen* for the HPC course **Programming Skills** a few good features can be made about the run times of said program. Additionally, it was possible to know what functions should be optimized in order to do a significant impact on the program's run time.

The first feature of the code that was studied was how the program's run time increases as the land-to-water percentage is increased. A linear relation between these was found, confirming that the computation time increases when the percentage of land in a fixed size file is increased.

Afterwards, it was shown that the program's run time also grows linearly when the input landscape file's number of land/water squares is increased linearly. This confirmed the intuition that the most important factor for the program's run time is the landscape's area.

Finally, the profiling part of the tests provided information as to what functions could be optimized in order to decrease the run-time of the program. Said optimizations should be done for writing the `.ppm` file, since this always represents most of the run time regardless of input file size (the percentage of the run time that it represents should also grow linearly).

It can be concluded that performance testing and profiling are important to know what are the weaknesses and the strengths of large, or complex, codes that cannot be studied by simply going through them. These methods are also very useful to know what sections of a code is taking up most of the run time and, thus, best suited for optimization.