Winter 2024

COSC 4P82 GP: Assignment 2

Due date: 9:00am Monday March 25, 2022. **No lates. No extensions.**
Instructor: B. Ross

Hand in: A report about your experiments (see assignment 1 for content and description). Electronic copies of your report, GP code, data, and results. Use Excel (or similar app) to create performance graphs, to be included in your report. You can work solo or in pairs.

System: Any GP system of your choice. You will require image file I/O in option B.

NOTE: Major mark deduction if ChatGPT or other AI LLM used to write any portion of report.

Select **only one** of (A) Predator-Prey, or (B) Evo-Art and procedural textures.

A. Predator-Prey (Pursuit)

Review of Artificial Ant Problem: During lectures, the "artificial ant" was discussed. The problem involves evolving an ant "brain" that can follow a trail of food. The language was very basic:

- MOVE: move forward
- LEFT: Turn left
- RIGHT: Turn right
- IF-FOOD-AHEAD A B: If food is in the cell in front of ant, execute A; else execute B
- PROGN2 A B: execute A, then execute B.
- PROGN3 A B C: Execute A, then B, then C.

Note that the simulation has the following properties:

- A fixed trail is used. It has a set pattern of trails and food. It is defined as a text file. It is refreshed before executing every simulation (fitness calls). Ants always start at one fixed position at the start of the trail.
- Fitness is the number of food eaten on the trail.
- When an ant eats the maximum food on the trail, the GP run can stop, since no ant can do better than that.
- A "trace" of ant movement is recorded in a 2D character array. It is dumped at the end to show the ant behaviour.

Predator-prey: This question involves modifying the artificial ant into a predator-prey application. There is one predator, and multiple prey. The fitness goal is that the predator eats as many prey as possible in the simulation time allowed. Prey are essentially 'dumb' pre-programmed food that move around at random.

To do this question, you need to implement the following.

A. **GP Language**: It is important that your GP language (functions, terminals) is sufficient to allow a good solution to be evolved. The predator should react to basic information from the simulation environment, for example, changing direction towards a prey (and/or closest prey). Thus, a special IfPreyBehind (Ahead/Left/Right...) function (similar to IfFoodAhead) could be included. PROGN2 will be useful too. But you may include others –be creative!

B. **Fitness**: The number of prey eaten.

C. **Simulation**: This is not necessarily complicated. Your simulation needs to update and record the state of all entities (predator, prey) – location, current predator direction, time count, and any other relevant factors. A predator's state changes according to the functions and terminals executed in its tree (just like the artificial ant). Prey is pre-programmed. Start with simple prey, and only make them smarter once simple cases have been studied (advanced prey can be comparative experiments).

D. **Output:** Make sure you can save enough information to understand what happened during a given simulation execution. A simple "trace image" can be created (out of ascii). However, for long simulations with lots of predator and prey movements, this might become complicated to look at, and so you might reduce it to only "predator movements" , "locations where prey eaten", and "location of live prey at end of simulation" (special ascii characters for all). A more ambitious approach would be to generate a trace file of all simulation entities, and then replay it in an animation player. Screen capturing can be used to make a video. Other than when debugging, you shouldn't normally see the simulation during the GP run – too slow!

E. **More output:** Also generate statistics of fitness (popn, best), tree size, etc.

F. **Multiple runs:** As in assignment 1B, run 10 runs per experiment, all using different random number seeds.

G. **Comparative experiments:** Try a few variations of your simulation –different GP language components, or different simulation rules, etc.. For example, you might want to compare wrap-around arenas with arenas with surrounding walls that cannot be crossed.

Testing your solutions: You should also run your best performing solution in each run against a set of new simulation instances, each with different starting predator locations. I recommend running your best solution 10 times, and recording statistics about each test performed. Note that sometimes an individual can be designated a "best solution" (high fitness) based on an accidentally good simulation. Therefore, testing it later with multiple simulations can verify whether it is truly a good performer or not.

Important technical consideration for GP language: You want the prey to move around when the predator moves. However, due to the way GP tree execution works, a predator can do a large number of steps in one tree execution. In such cases, the predator might make 20 plus moves, and when done, you'd then update the prey. A predator might eat a dozen prey before any had a chance to move! This would naturally bias fitness to favour predators with huge trees!

Therefore, you have to update all the prey after each function call in the tree that changes the state of the predator. For example, after moving the predator one cell, immediately call a method that updates all the prey. If the predator turns direction, again update all the prey. This gives prey a fighting chance, and keeps fitness unbiased by tree size.

Miscellaneous advice:

- Be sure to reset <u>all</u> the simulation variables before each new simulation.
- Make sure your GP language functions and terminals are not too powerful! It is easy to pre-program highly intelligent strategies into a function, so that GP has nothing to evolve. The goal is to give the GP language an adequate set of basic building blocks that can be used to evolve intelligent strategies. This is called *emergent intelligence* or *emergent behaviour*.

- Start with dumb prey. Perhaps they randomly move up, down, left, right, or stay stationary per clock tick. If you have time, smarten them up for a comparative experiment.
- Maybe you can add "mines" to the arena, or poisonous prey, which predators should not tread upon or eat, or else they die (and get bad fitness scores!). You will need danger sensor functions or terminals.
- Creativity is encouraged!
- Project idea: Co-evolution of predators and prey.
- Project idea: Teams of cooperating predators.

B. Evolutionary Art and Procedural Textures

This question involves implementing an evo-art system with GP. A colour image on a monitor or image file consists of a grid of pixels. Each pixel has red, green, and blue values. A procedural texture is a way to mathematically compute each pixel's colour. Here is an example of a procedural texture for a pixel at location (X, Y):

```
Pixel_Red(I, J) = sin(X)*cos(Y)
Pixel_Green(I, J) =  X/2 * 5/Y
Pixel_Blue(I, J) = 0.95
```

where (I, J) are the bitmap indices, and (X, Y) are the texture space coordinates. Note that every coordinate (X, Y) has a colour defined for it. Note that (I, J) are usually integers, while (X, Y) are normally floating point values; typically, both are different. The idea is that the formulae define a texture space on the infinite XY plane, and each (X, Y) coordinate has a colour defined for it with the equations. If the X and Y coordinates do not appear anywhere in an expression (see the Blue equation above), then that colour channel will be constant for all pixels.

You are going to let GP evolve the formulae for R, G, and B. This can be done by defining a floating point language, in which X and Y are terminals that are assigned the texture coordinates that map to the particular pixel (I, J) being rendered. You should have a root note in the tree that has 3 arguments -- each being a for R, G and B. Therefore, to render a bitmap:

```
For I = 0 to Max_Row {
        Y = (texture coordinate for I);   // Y is a terminal in GP language
        For J = 0 to Max_Col {
                X = (texture X coordinate for J);  //X is also a terminal
                Pixel[I, J].red  = evaluate_tree(tree.arg[0]);
                Pixel[I, J[.green = evaluate_tree(tree.arg[1]);
                Pixel[I, J].blue = evaluate_tree(tree.arg[2]);
        }
}
```

You are recommended to convert the results from the "evaluate_tree" calls into floats between 0.0 and 1.0. This is a standard data range for many image files. (Integers between 0 and 255 are also possible). Therefore, make each value positive with "absolute value", and truncate values between the minimum and maximum of the data type you are using.

Note that the texture coordinates reside "on top of" the pixel bitmap. Since texture coordinates are conceptually infinite in X and Y (it's a plane), the bitmap is sampling a small sub-area of this plane. You should indicate the minimum and maximum X and Y that you will use to place on top of the bitmap. Put these values in the parameter/setup file.

But before you do RGB colour: First do the problem using greyscale (black and white) images. Greyscale means that the R, G, and B values are the same for each pixel. Hence you only need one 2D arryay to save the pixels, and a single expression tree to compute greyscale values. Do a distance match using a greyscale target image. Distance calculations will be simple too (see below). Once you have done an experiment with greyscale, then expand it to full RGB. (If you think about it, the RGB colour version is like doing 3 greyscales simultaneously, as each R, G, and B is equivalent to the greyscale problem).

Fitness: In order to let GP automatically evolve a texture, there needs to be some sort of fitness criteria. You will use a "colour distance" calculation for the fitness. This involves reading in a colour image at the start of a run, which will be a "target" to match with each population member's rendered image. This image should be a small size, e.g. 128 by 128 pixels, because it is very expensive computing the texture for a lot of pixels. Assume that the target image is represented in a similar format as Pixel[I, J] above, and has the same size as Pixel. Then once the Pixel array is filled in as shown in the above pseudocode, the distance is computed by:

```
Dist = 0.0;
For I = 0 to Max_Row {
        For J = 0 to Max_Col {
                Dist_Red = (Pixel[I,J].Red - Target[I, J].Red);
                Dist_Green = (Pixel[I,J].Green - Target[I, J].Green);
                Dist_Blue = (Pixel[I,J].Blue - Target[I, J].Blue);
                Dist = Dist+sqrt ( DistRed*DistRed + DistGreen*DistGreen +  DistBlue*DistBlue );
        }
}
```

A version for greyscale:

```
Dist = 0.0;
For I = 0 to Max_Row {
        For J = 0 to Max_Col {
                Dist_Grey = (Pixel[I,J].Grey - Target[I, J].Grey);
                Dist = Dist + DistGrey*DistGrey;
        }
}
```

Note that if the GP image in Pixel is an exact match to the greyscale or colour target image, then the overall distance is zero. However, this is **very unlikely** to happen for anything but the most basic target images.

GP Language: A basic texture language can be the same as assignment 1 (a), except that both X and Y are terminals. Include trig, log, exponent, and others.

You should also include at least one of the following (the more you include, the more interesting images will be... but the longer it will take to execute the trees!). I will include C-source code for a bunch of them; please feel free to port one or more of them into your own system. If you find some

online already in your system's language (Java, Python), feel free to use them –as usual, **cite your sources!**

- Perlin noise
- Turbulence
- Mandelbrot/Julia set (warning: very cool but **very** computationally expensive)
- Some other interesting noise-based texture function.

Efficiency: Procedural textures can be very expensive to evolve. One colour image involves executing 3 trees (R, G, and B subtrees) on each pixel. A 256 by 256 bitmap will require 196,608 subtree executions for a single fitness evaluation! Therefore, a practical requirement is that image sizes for fitness evaluation are small. 128 by 128 has 25% the area as 256 by 256. On the other hand, a small thumbnail is not very impressive as art. Therefore, you should set sizes for a high-resolution "final output" image at the end of each run, which will output a large-sized PNG file of the best solution obtained, for example, 2048 by 2048. (You don't need a target image of that size, since the best solution expression doesn't need the target to render a hi-rez output -- the target image is only used for fitness).

Another optimization idea: Imagine a checkerboard were placed on the thumbnail pixels. Then you should only render the pixels beneath black squares of the checkerboard, and find the fitness distances for only these pixels. This cuts down processing by a half, with minimal effect on fitness accuracy. For a 128 by 128 thumbnail, this results in 24576 tree executions per individual (8 times faster than the 196,608 mentioned above for 256 by 256 images!).

Miscellaneous advice:

- You should compare a few different GP strategies and options, for example, different GP parameters, target images, or GP languages. Compare them in the report.
- Population sizes will be on the small side (300?).
- Note that you will never reproduce the target image (unless it is very basic). Consider the target image as being a "composition guide" or "inspiration" for evolution.
- You might want a debug mode in which each population member's rendered image is saved as a PNG file. Make sure to name the image files with the generation and ID number, so that they don't clobber each other!
- Project idea: Texture generation screams for GPU acceleration (CUDA programming).
- Project idea: Interactive evolution: Using a GUI, a human user behaves like a fitness function, scoring thumbnail textures from 1 to 10 (best).

References:

Java image I/O: http://java.sun.com/products/java-media/jai/iio.html