SNAKE AI

James Ball

Contents

Analysis

Outline of problem:

Students can find it hard to get a grasp of how neural networks and genetic algorithms (GAs) work together to create Artificial Intelligence (AI) as this can be a complicated topic. It can be explained at a basic level quickly but getting a detailed understanding of the subject can be more difficult.

To help explain how neural networks (NNs) and genetic algorithms can be used, I propose a program that implements these to 'learn' how to play the classic game Snake. This will be a visual way of seeing how the AI progressively teaches itself to play the game, hopefully making it clearer how this is possible in comparison to being taught the theory.

In addition to learning how to play Snake, my program should display a graph detailing how the snakes have improved in score over time. This will visualise how the snakes have progressively improved at the game, demonstrating to students the learning that is occurring over time. The program should also visualise the neural network of the snake as this will help explain what neural networks are made from and how they work.

Users of program:

Primary users of the program are students studying neural networks and genetic algorithms within computer science. Other users include hobbyist programmers interested in AI and wanting to get a greater understanding of how these subjects work.

I will be discussing ideas and receiving feedback from two users. The first user is an undergraduate computer science student that is researching neural networks and genetic algorithms and the second is a younger student that is interested in AI and wants an introduction to genetic algorithms and neural networks.

Initial focus group with users:

I held an initial focus group with the two users detailed above to discuss program ideas, goals and features. Firstly, I explained that I am looking to create a program that will learn how to play Snake to help explain how neural networks and genetic algorithms work. I then asked some questions and proposed ideas about the program to see their feedback.

The main points from the group were:

- It can be confusing to understand how programs can learn how to play a game or 'learn' in general. They thought that seeing a program go through this process would help to explain how it is possible.
- The younger student said he struggles to understand how neural networks work and agrees that seeing the program learn could help him grasp the concept.
- The undergraduate student said that graphs should show all the key points where the AI has learnt something new, and as a result, has scored much higher.
- They also thought only the best snake's neural network should be drawn so we can get an idea of how the AI works.
- They loved the idea of visualising neural networks and think it will allow them to work out how exactly neural networks work.

- The undergraduate student mentioned that he thinks it would be good to include as much information as possible about the program as it runs, such as the maximum score achieved, average score of all snakes over time, current score of the best snake and how many snakes have already died, just as a few examples. He suggests that the more information there is, the more it will let students understand what is happening as the program runs.

Overall, both users thought that the program would be a great educational tool for visualising how an AI can learn how to do a task and explaining how this happens more clearly.

Research of algorithmic methods:

Genetic Algorithm:

A Genetic Algorithm (GA) is an Evolutionary Algorithm that tries to find a solution to a problem using processes inspired by natural selection within biological evolution. Some information sourced from https://en.wikipedia.org/wiki/Genetic_algorithm and https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm).

An initial population of random 'genomes' are created. The size of this population depends on the application of the GA. These genomes will try to solve the problem, and they will be rated on their ability to solve it after a predefined time or once they have 'died'. This rating is known as the fitness of the genome. For example, the length of the snake is the fitness for this program.

A process called selection chooses the best performing genomes for breeding into the next generation. Genomes that have performed better are more likely to be selected to survive into the next generation. For each genome in the new generation, two parent genomes are selected using this process and are bred in a process called crossover. Crossover is like biological breeding; attributes from each parent can be seen in the child genome. There are various crossover methods including uniform crossover where each feature of the child genome is randomly chosen from either parent (i.e. 50% of features are represented by one parent, 50% by the other).

Once a child genome has been produced through crossover, it is mutated to slightly modify its features. Mutation can involve partially changing a random number of features in the genome or completely overwriting a certain feature with a random new feature. Mutation and crossover introduce natural variation in the population with the hope that this variation will increase the fitness of the genome.

This process of creating a child is repeated for the rest of the new population. This should create a new population that consists of genomes that share features of the best performing genomes in the last generation, leading to a gradual improvement in fitness from the population.

Each time a population has finished and has been rated, these processes repeat.

Genomes trained using a GA aim to reach high-quality solutions after a number of generations through this gradual 'learning' procedure. In summary, GAs train genomes to reach as high a fitness as possible through a rating procedure, selection, crossover and mutation.

Research of algorithmic methods:

Neural network:

An (artificial) neural network (NN or ANN) is an interconnected network of nodes or 'neurons' that is inspired by biological neural networks which compose the brains of animals. Each node represents a neuron within the brain and each connection represents a synapse. Nodes have an activation function applied which places inputs within a range of 0 to 1 to normalise the results. Each synapse has a weight associated with it (ranging from -1 to 1) which scales the values input and outputs this scaled value.

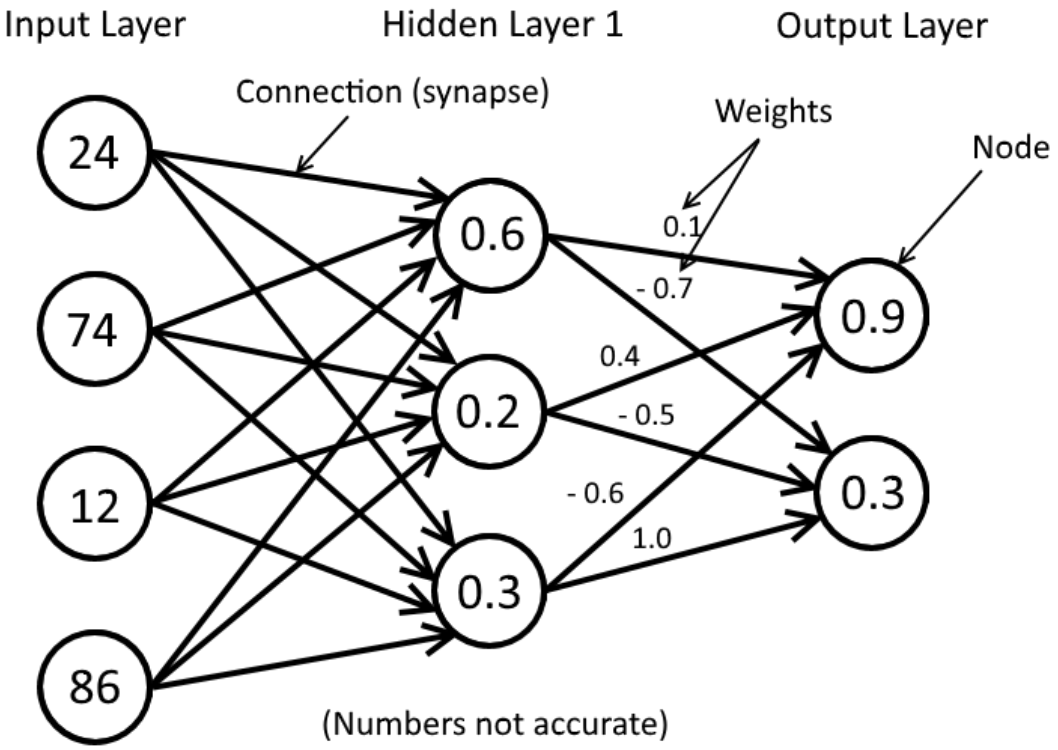


Figure 1 – Neural Network diagram detailing connections, nodes, weights and layers.

Nodes/neurons in an ANN are arranged in layers, where values are passed from one layer to another. ANNs have a single input layer and output layer and any number of hidden layers. Inputs are loaded into the input layer and their values are weighted by the connections within the ANN and passed to the next layer. This process of feeding values through the network is called feeding-forward. Usually each layer has a bias node, which is a node with a fixed output of 1.0 which is then weighted to affect results in the next layer.

Parameters such as the number of nodes in each layer and the number of hidden layers is determined by the complexity of the problem and experimenting to find the most effective parameters. These are known as hyper-parameters.

Weights between layers manipulate the input to produce an output array which, once an activation function has been applied, will consist of decimal numbers between 0 and 1 (if sigmoid is used; other activation functions can lie in other ranges). Outputs are normally utilised by carrying out an action based on the highest value in the output array. The closer a node is to 1, the more confidence it has in the node it has selected. For example, in figure 1, the action associated with the top node in the

output layer would be carried out as it has an output of 0.9 which is greater than 0.3 and is therefore more confident.

The process of 'training' a neural network to solve a problem involves reaching the correct weights so that the right output is reached when data is input. This can be achieved in a variety of ways, but this program will use a GA as discussed above. Genomes that perform better are more likely to reproduce, so their weights will be passed into the next generation.

Neural networks are represented using weight matrices, which will be a 2D array in my program.

Matrix (2D Array):

Matrices are two-dimensional vectors that can be represented by the two-dimensional array data structure. Neural networks consist of weight matrices, hence why implementing matrices is required in my program.

Although 2D arrays are easily explained as data elements organised in rows and columns, matrices are more complicated because of the mathematical operations that can be applied. They must support multiplication between other matrices to be used in a neural network.

Matrix multiplication is defined as follows (source: https://en.wikipedia.org/wiki/Matrix_multiplication):

If $\mathbf{A}$ is an $n \times m$ matrix and $\mathbf{B}$ is an $m \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{pmatrix}$$

the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $n \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj},$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, p$.

Figure 2 - Matrix multiplication

This means that the *ij*th element of **C** is found my multiplying each element in the *i*th row in **A** by the respective element in the *j*th column in **B** and summing all these individual multiplications.

Implementation of this multiplication is required for generating an output in my neural network. I will implement a Matrix as its own class so that I can use it as its own data structure.

Research of required data structures:

Lists:

I will make use of lists in my program to store the sections that make up the snake's body and datapoints on the graph. Java, the programming language I am using for this project, has a native list data structure called ArrayList, which is what I am using for the lists in my program.

ArrayLists have a range of functions that I will utilise in my program:

- .add(Element e) – This adds element 'e' to the list. This is an arbitrary variable that can be any data type.
- .addAll(ArrayList l) – This adds all values of one list to another.
- .get(int i) – This returns the element at index i of the list.
- .size() – This returns the length of the list.
- .remove(int i) – This removes the element at index i from the list.
- .toArray() – This returns an array of the data type that the list is made from. All elements in the list are elements in the returned array.

Vector:

Vectors will be used to store the current direction the snake is moving in and the position of the snake. Processing, the IDE I will use, includes some libraries on top of the native Java libraries. This includes a class for vectors called PVector.

PVectors have a range of useful functions for manipulating vectors:

- .add(PVector p) – This adds one PVector to another by adding each dimension together (i.e. x, y, z)
- .sub(PVector p) – This subtracts one PVector to another.
- .mult(PVector p) – This either multiplies a PVector by a scalar or by another PVector.
- .div(PVector p) – This is the same as .mult(), however it divides instead.
- dist(PVector p, PVector v) – This finds the distance between two vectors.
- .limit(float f) – This limits the magnitude of the PVector by a float.

Usage of computational methods:

Abstraction:

The GUI of the program will be heavily abstracted to improve the performance of the program and make it easier to understand.

Genetic algorithms use a population of 'genomes' which will be my snakes. Instead of showing every single snake at the same time, I will only show the top few snakes as these are the most interesting to watch, as they represent the best performing snakes of the population. Additionally, only the best performing snake's neural network should be visualised. It is unnecessary to display the neural network of more than one snake as neural networks across a population will be very similar. Both tweaks would greatly improve the performance of the program, as well as its usability, as they remove lots of unnecessary detail.

Genetic algorithm processes such as selection, mutation and crossover will not be mentioned in the GUI at all, as these are behind-the-scenes processes that are not necessary to cover as they do not help to teach how AI can learn over time. It is assumed that the user would already know these processes and is using this program to see how they work in practice.

The snake that is currently doing the best should be a different colour to make it clear to the user which snake is doing the best. This highlights the important details of the program to the user.

Visualisation:

My program will make heavy use of visualisation as this is essential in showing how the AI learns over time.

The snakes that are still alive will be visualised in a grid, which is the area in which they can move. This allows the user to see how well the snakes are doing in the current generation. Without this, the user would not see how the AI learns different techniques to get a higher score.

The average score and highest score of the population each generation should both be visualised in two separate graphs. This will let the users see at which point the snakes learnt something new and therefore got much better at playing the game. It will explain the idea in genetic learning that once a single 'genome' learns something new, this newly acquired knowledge will quickly spread to the rest of the genomes. The inclusion of these graphs would also help diagnose points at which learning has slowed and the AI struggles to learn anything else.

The neural network of the best snake should be visualised, detailing all the weights, connections and layers of the neural network. This is very helpful for the users in showing how neural networks work. Connections should appear as different colours depending on the strength of their weight. Without this, it would be unclear as to how the AI works. It gives the user an idea of what is controlling the AI to move.

Decomposition:

There are many key sections to this program due to the complexity of it. It can be broken down into 4 main areas, which can then be broken down further.

Each prototype begins with a decomposition diagram, describing how the prototype is to be broken down.

Creating a playable Snake game:

Before any AI can learn to play Snake, I must recreate it in my own program. Snake can be broken down further into a few sections:

Collision detection should be implemented so that the snake dies when it hits its tail or a wall. A movement system must also be created so that the snake can correctly move around the grid, regardless of the length of the snake. Finally, I need to implement an apple for the snake to collect, along with a system that will grow the snake's tail and count its score.

Implementing a framework for creating and modifying neural networks:

I need to implement a neural network data structure that can be used by the snakes to move. This must be created before the genetic algorithm, as the GA modifies the neural networks of snakes. The specifics behind neural networks is covered in detail in the 'research of algorithmic methods and data structures' section.

Firstly, a matrix class must be implemented which allows for multiplication, adding bias nodes etc. Using this class, a neural network class should be implemented that allows for feeding forward, creating neural networks of differing sizes etc. I would also need to create a Player class that extends the neural network class which will control the movement of snakes using their neural network.

Computational needs:

Implementing a genetic algorithm:

To allow the snakes to learn how to play the game, I need to implement a genetic algorithm. This is detailed in my research, but the decomposition of this section is mentioned here.

I need to implement crossover between different snake's neural networks, mutation of snake's neural networks and selection of the best performing snakes from the previous generation. These will all be significant functions that can be coded in any order. I would also need a main function that carries out all these processes to create a new generation of snakes.

Visualising progress:

To show the process of the snakes learning, it is important that the fitness of the snakes is visualised, so the user can understand how genetic learning works more easily.

Visualisation involves creating all the fundamental code for graphing the data as well as a function to visualise the neural network of the best snake.

Pipelining:

Each frame, lots of data is pipelined through the program. Firstly, once all the snakes have been generated, the 'vision' of each snake is passed through the neural network for that snake and the output decides what direction the snake moves in next. Once the snake has moved, the grid will be updated to represent this, and the grid will then be visualised, so the user can see where the snake is.

At the end of each generation, when all snakes have died, the data of all these snakes is processed by the genetic algorithm, resulting in a new generation of snakes that have learnt from the last. The snakes then move according to their 'vision' once again and the process continues.

After each generation, data will also be added to an array of average score and max score which will then be visualised in a graph as mentioned above.

Computational needs:

Required inputs and outputs:

Inputs:

As the program is a simulation of a genetic algorithm, no user input is required for the program to run. The program will begin and continue indefinitely once opened. During development, inputs such as loading neural networks from previous executions of the program may be useful, but the program initially doesn't require any user inputs.

Inputs that will need to be tested and compared will be the hyper-parameters (detailed in the neural network research) of the neural network. Although these parameters might have a sensible range of values, lots of trial and error is required to find the values that are best suited for the snakes to learn. Hyper-parameters that will need tweaking include the mutation rate of weights in the neural network and the structure of the neural network (number of layers and nodes). The size of the population of snakes should also be compared for maximum efficiency.

The snakes will also be given inputs so that they can understand the environment they are moving around in and can work out the direction of food, walls and their tail. It is not immediately clear what the inputs should be for the snake, as they should be as minimal as possible whilst providing enough information for the snakes to develop advanced techniques for playing the game. Experimentation of the snake's inputs will be required.

Computational needs:

Outputs:

Outputs required include the grid that the snakes will be moving on, along with the apples they must eat, and the graphs that will be displayed to highlight the development of the snakes in learning how to play the game. The data required for these graphs is the average score of the snakes along with the fitness of the best snake. A neural network graph will also be shown; highlighting the structure of the snake's brain. All data for this is already part of the neural network, so no calculations must be made.

Calculations during runtime:

Genetic algorithms are very computationally intensive, as is common with AI. Many operations will occur every frame, so the time they take must be minimised if possible.

Real-time calculations:

Inputs are given to each snake every frame so that an output can be generated using their neural network. The calculations made here must be minimal as hundreds of outputs will be generated each frame to determine the next move for each snake in the population.

The GUI elements that are rendered each frame will also be computationally intensive, such as the snakes, graphs and neural network diagrams. I must be careful, specifically with graphs, that the performance of the program doesn't degrade over time as the number of data items in the graph increase. Although displaying all data points in a graph will be at least an O(n) operation, I may be able to reduce the number of data points as a form of abstraction to reduce the computational requirement. I must be sure that I am not rendering more than what is necessary, as this will dramatically reduce the performance of the program.

Longer-running calculations:

Any calculation made between generations and at the start of the program can last much longer than those running each frame. Execution of a current generation takes far longer than the execution between generations, so more processing between generations will not have a big impact on the overall performance of the program.

These types of calculation include the crossover, mutation, selection and the resetting of genomes. No rendering is required for these calculations so the time that these take to execute shouldn't be substantial.

File storage requirements:

File storage will be required in prototype 3 as I aim to include a neural network saving and loading feature in which high-performing snakes can be saved and loaded in future executions of the program. This requires me to process all of the features of the snake's neural network and save these into a readable format. The file format I will use for this is .JSON or JavaScript Object Notation, as although this is intended for webpages, it proves a popular format for processing data, and it is supported as a default library by Processing; the IDE and language I am using. This will readily allow me to convert each attribute of the neural network into a format that can then be loaded and understood by my program in future executions.

Documentation for this functionality can be found at processing.org/reference/JSONObject.html, and the main functions I will make use of are getFloat(), getJSONObject(), getJSONArray(), setFloat(), setJSONObject() and setJSONArray(). As well as saveJSONObject() and loadJSONObject() for saving and loading from files. These methods make it easy to implement a system in which I can save and load neural networks in my program.

Hardware and software requirements:

As previously mentioned, the IDE and programming language I am using for development is Processing. This is a modified version of Java that includes some pre-built libraries and additional features as standard. I am not using any further libraries, other than those included by Processing and some standard Java libraries. This means that the only requirement for developing the program is the Processing IDE, along with an updated version of Java (will already be installed on most computers) along with a standard computer with adequate memory (4GB+).

The hardware required for running the program once completed will be more significant. The speed at which the program runs is completely dependent on the performance of the computer it is run on. This means a performant CPU and graphics card are greatly beneficial to the performance of the program. Therefore, there is no strict hardware requirement, but the better the hardware, the quicker the snakes will learn.

Prototype Aims:

The success criteria and limitations in the following sections are a result of a second focus group held with the stakeholders. Much of the criteria was suggested by these stakeholders. Explanation for each criterion is also given.

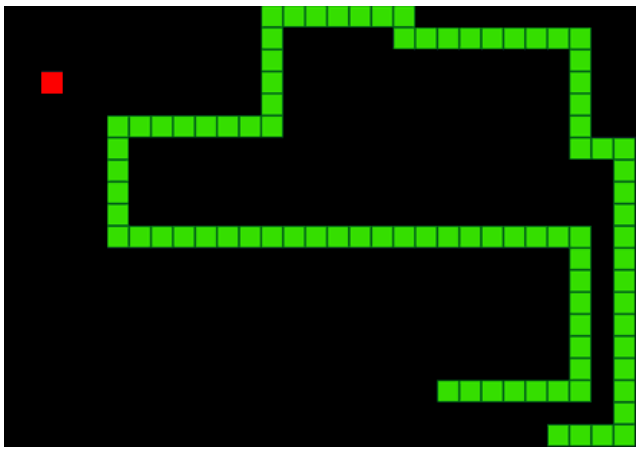
Prototype 1:

In the first prototype, I will recreate Snake in my own program. Although it is a simple game, a prototype is useful to ensure I have the infrastructure to begin implementing the genetic algorithm once it is complete.

Limitations:

Due to the simplicity of Snake, there are not many limitations to be concerned with in this initial prototype. The main limitation I should be conscious of is the complexity of rendering the snake's position along with the apple to a grid. This process would be repeated with hundreds of other snakes each frame, so I am limited on the number of snakes I can render at once. I should also be aware of how the genetic algorithm will be implemented; coding the Snake game in a way that makes it easily accessible by the GA. This could prove to be difficult as the GA is not yet implemented.

Detailed graphics for this game are far beyond the scope of this prototype and are not beneficial for the project's goal. Because of this, enlarged pixels will be used to represent the snake and apple; reminiscent of classic Snake:



Prototype Aims:

Success criteria:

The overall goal for this prototype is to replicate the classic game of Snake in a way that facilitates a genetic algorithm being added in the future. This will be achieved as follows:

1. Create a collision system for the snake.

   This system will result in termination or resetting of the snake if they hit the walls or their tail. Without this criterion, Snake would be unfinished and there would be no way of losing.

2. Allow the snake to grow and turn, as presented in the image above.

   Growing makes Snake increasingly hard as the snake gets longer due to the size of the snake taking up the screen. It is important this is implemented as it raises the skill required to get a high score, which will demonstrate how an AI can learn in more detail. Although turning may be easy to implement, having the snake's tail bend is more challenging and this is required to have a fully implemented game of Snake.

3. Implement a solution to rendering the snake that can be adapted to display multiple snakes in future prototypes.

   As mentioned in the computational needs, it is very important that multiple snakes can be rendered at the same time efficiently. If this criterion is met, this efficiency can be more easily achieved. Without meeting this, I would have to reimplement the rendering system to support more than one snake.

4. Allow the game to be easily controlled by a neural network, once implemented.

   It must be coded in a way in which I can easily change what controls the direction of the snake. It must be easy to change from keyboard control to neural network control for compatibility with future prototypes.

5. Allow for a variable-sized grid for the snake to move on.

   If the grid size can be chosen, I can demonstrate the effects of using the same Snake AI on different grids to the end user. This benefits the goal of my program but a variable-sized grid is not essential in this first prototype.

6. Allow for both time-based movement (move every x seconds) and frame-based movement (move every frame).

   Time-based movement will be in use for this prototype as it makes sense for the snake to move every x seconds, however once I implement the genetic algorithm, the snakes will move as fast as possible, so they learn quicker. This means frame-based movement would be preferable, where the snake moves every frame. I must implement a solution that makes it easy to switch between the two.

Prototype 2:

The second prototype will implement the genetic algorithm. In this prototype, all classes for matrices and neural networks will be created as well as all the functions to crossover, mutate and select the most fit genomes.

Limitations:

I am limited by how advanced I decide the algorithm to be. I cannot have a population of snakes that is too large because it will slow down the program due to the number of snakes that are processed each frame. I am also limited by the size of each snake's neural network. The neural network cannot have too many layers as the added complexity will be increasingly demanding on the program. Generating an output is an $O(n^2)$ operation where n is the number of nodes, which is why this is an issue. There will exist a value for each parameter that is optimal; the complexity of the neural network will be balanced with the processing required, but it will require lots of trial and error to find these values.

Another limitation is the complexity of the learning process for the snakes. Other examples of evolutionary algorithms include NEAT or Neuroevolution of Augmenting Topologies (see http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf). This is a more advanced technique that evolves the structure and number of nodes in the snake's neural network as well as the weights in the neural network. Although this is a very effective algorithm, it would be unnecessary to implement because it goes far beyond a traditional genetic algorithm. This means it would not be as beneficial in teaching people how GAs work.

Success criteria:

The overall goal for this prototype is to allow the snake to learn how to play the game.

1. A system for choosing the best performing snakes randomly must be implemented.

   This system must ensure that although the best snakes are most likely to move to the next generation, there is also a chance that worse snakes will move to the next generation. This introduces some randomness seen in biological natural selection. This selection process is essential in allowing the snakes to learn as they must learn from the best performing snakes in the last generation.

2. Mutation must be introduced in each genome after each generation.

   Mutation introduces more randomness found in natural selection. It is important as it means that genomes can mutate and those that benefit from mutation will perform slightly better, leading to a gradual improvement of the whole population. Without mutation, genomes can get stuck at a point where they cannot improve any further because there is no more innovation. This point is known as a local maximum.

3. A system for crossover between two genomes should be implemented.

   Crossover between two different genomes can enable the best elements of two high-performing genomes to be present in the child genome. It is another element of randomness present in natural selection, along with mutation and selection. It also ensures that attributes of a high performing snake are transferred to other snakes quickly.

4. A method of easily creating neural networks of different sizes should be present.

   Changing the dimensions of the neural network shouldn't require any recoding. I should be able to set the dimensions of the neural network using an array of integers, defining the number of layers and the number of nodes in each layer. This would allow me to quickly test different neural network structures and find the most efficient structure for the problem. It also allows me to demonstrate how changes in structure dramatically impacts performance of the snake, supporting the purpose of the program.

5. I need to create conditions that determine the death of each snake.

   Snakes are likely to get stuck in repeatable movements where they will never eat an apple or progress any further. This is an issue because they will never hit themselves or a wall, so they will never 'die', but they will never progress either. These loops should be detected, and the snakes should be killed if they are stuck in these loops so that the program isn't stuck waiting.

Prototype 3:

The final prototype will introduce the monitoring of the population's progress. Graphs determining the average score of all snakes and the max score in a population should be shown so that progress is easily determinable. A way of exporting and importing already generated neural networks from previous executions of the program would also be desirable so that progress can continue after closing the program. This prototype is very important as it showcases the development of the AI over-time using a genetic algorithm to the user, which is the goal of this program.

Limitations:

There is a limited amount of computational power available to this prototype. The features added in this prototype will ideally not significantly affect the performance of the rest of the program as this doesn't directly contribute to the AI learning. The processing required for the graphs used should be minimal. This may prove to be difficult due to the complexity of rendering graphs with hundreds or thousands of datapoints.

The number of graphs that I can show at once is also limited. If many graphs are being rendered, it will be hard to fit them all on the same window and it will be very computationally intensive rendering them each frame. I should stick to a few important graphs to combat this.

As with prototype 1, advanced graphics are outside the scope of this project, so I will be limited to simple graphs that display progress over time. The graphical fidelity of the graph isn't very important; it only needs to display the data correctly.

Success criteria:

1. Graphs must be generalised to fit a variety of data and labels (i.e. not specific to a single situation).

   Graphs that can easily be adapted for new data are required as I will have multiple graphs displaying different data. A system in which I can create new graphs with different data quickly is beneficial as it provides a layer of abstraction in my code which means I can focus on the data and labels of the graph, rather than how the graph works, once the graph structure has been created. This structure would be implemented using a class.

Prototype Aims:

2. Graphs must be simplified as the number of data items increase.

   As more data is added to a graph, more datapoints and lines will be rendered each frame. More processing is therefore required, which can reduce the performance of the program overall. A system must be implemented to reduce the number of data points once it reaches a certain point to reduce the processing power required. Other methods of reducing graph complexity would also be hugely beneficial.

3. Neural network graphs should show all connections between nodes and layers and visualise their weights.

   All connections between nodes should be visualised as lines between nodes and layers. The weights of these connections should be further visualised by a different colour or thickness of the line depending on the weight. Introducing this visualisation will allow students learning about genetic algorithms and AI to see how the weights change overtime and see how patterns appear in the weights as the AI develops.

4. The current state of the snake's neural network should be able to be saved and loaded so that a known-working snake can be loaded in future executions of the program.

   It should be possible to press a button and have the current best-performing snake's neural network to be saved in a readable format so this can be loaded and demonstrated to students in future executions of the program. This also means that any progress in the AI's ability can be saved so there is no requirement to restart the learning process.

Design, Development and Evaluation

Design Overview:

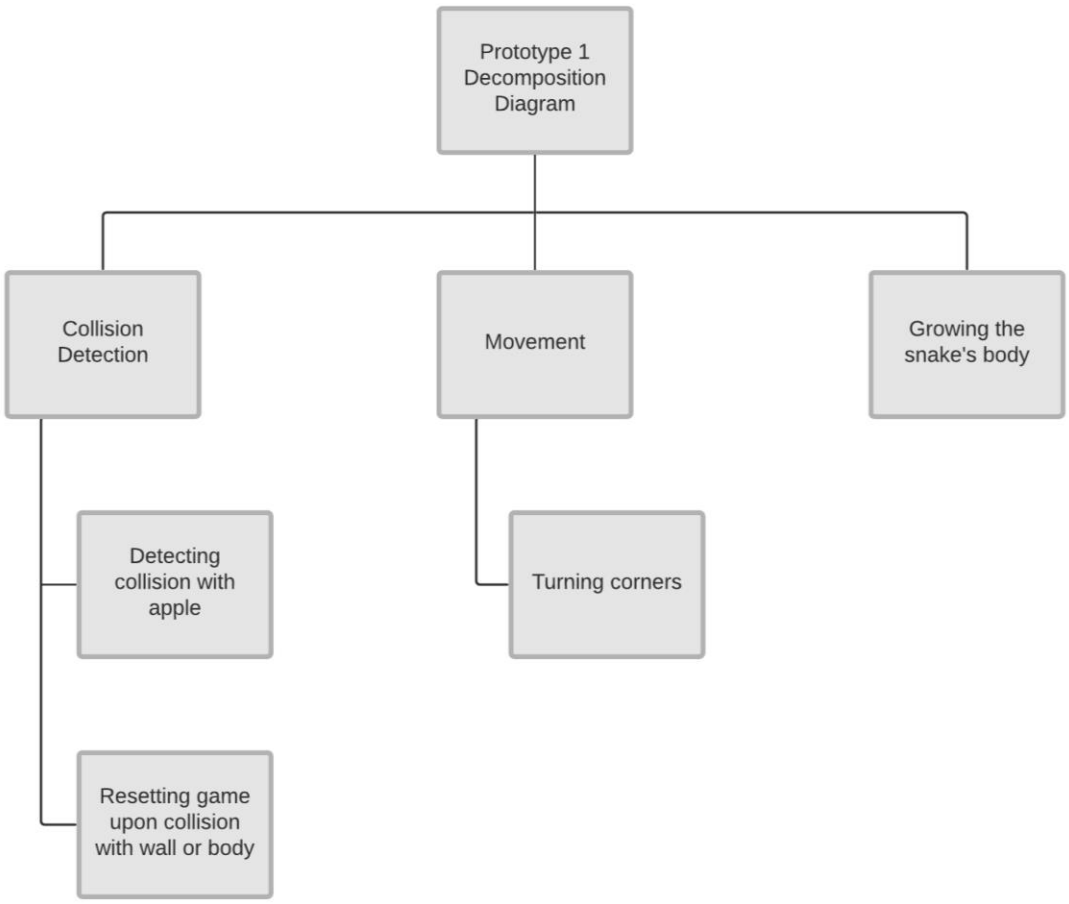
The programming language I will be using is called Processing (processing.org) which is a language and Integrated Development Environment (IDE) designed to easily visualise data that acts as a layer on top of the Java programming language. It is very powerful at creating visual demos and interactive programs, making it heavily suited for this program. I will explain differences in this language to Java as they appear.

Development and design will be separated into three prototypes as detailed in the analysis section.

Prototype One:

Prototype one will recreate Snake so that I can manipulate the functionality of Snake when creating my other prototypes. I should consider how the genetic algorithm will be implemented during the development of this Snake clone, so this code can be easily adapted in future prototypes.

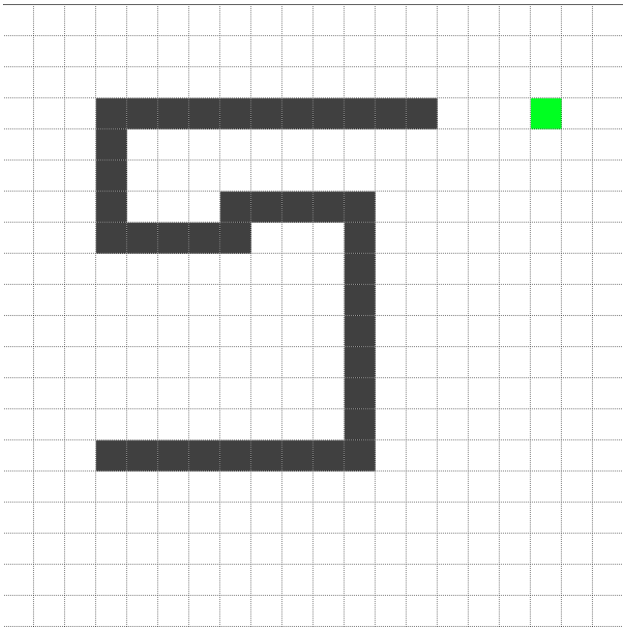
Decomposition Diagram:



This diagram shows the sub-problems I am breaking this prototype down into. There are three main sections to this prototype, the collision detection, movement of the snake and growing the snake's body. Collision and movement can be separated into further sub-problems, as demonstrated on the diagram.

Prototype One:

Initial Sketch:

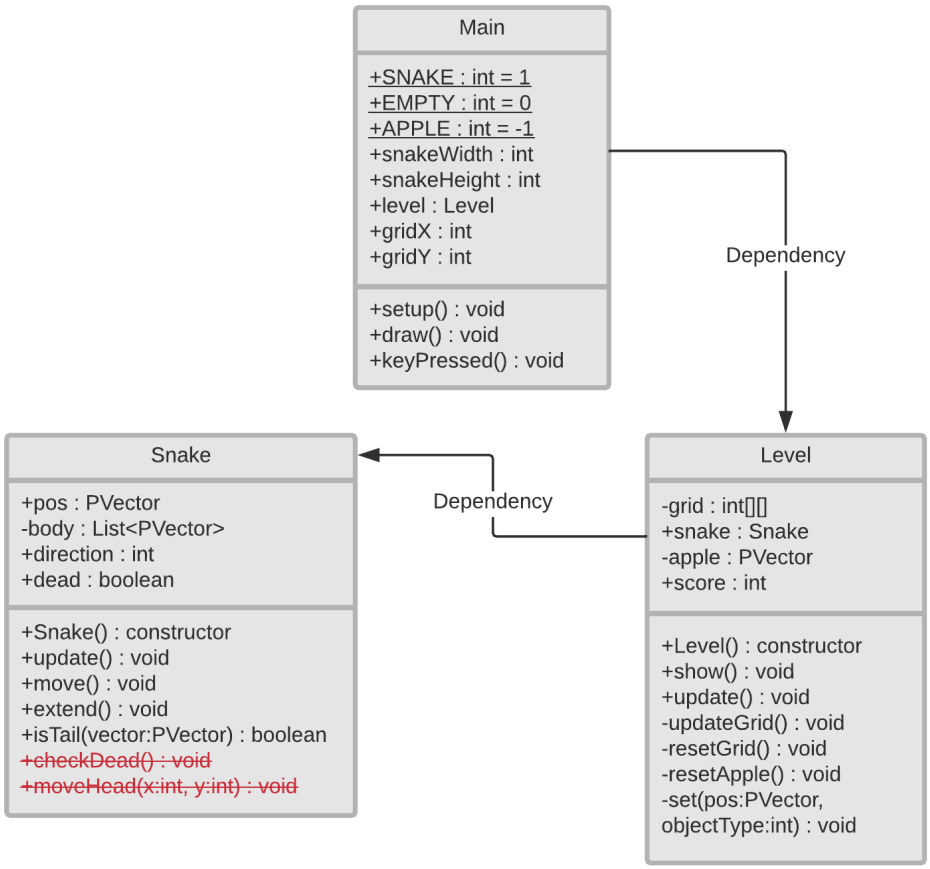


I have made a sketch to demonstrate what the first prototype should look like i.e. a classic Snake game. The dark grey line is the snake which will be able to turn and create bends as it turns. The green apple will increase the length of the snake and the score of the player by one each time.

The snake's body will consist of a list of vectors, corresponding to coordinates on the grid. When the snake moves, the next position to move to will be added to the front of this list and the tail element of the list will be removed.

Classes:

Class Diagram:



Methods that are red and crossed-out were removed from the prototype at some stage in the development log.

James Ball

Prototype One:

Explanation of key algorithms:

Below is a collection of explanations and pseudocode about the fundamental algorithms from all classes in this prototype. Algorithms not highlighted here are trivial and can be easily understood via code inspection.

Level.show():

This is the algorithm that displays the current state of the snake game. The current position of the snake, along with the position of its body parts, and the position of the apple are all visualised using this algorithm. This makes heavy use of the functions built into Processing, such as rect(), line() and fill() which draw a rectangle, draw a line and change the colour of drawn elements respectively.

Pseudocode for this method is below:

```
procedure show()
    // These define the dimensions of each square in the grid in pixels.
    // gridWidth/Height is the size, in pixels, of the visualised grid.
    // gridX/Y is the dimensions of the grid array.
    squareWidth = gridWidth / gridX
    squareHeight = gridHeight / gridY

    for (i = 0, i < gridX, i++)
      for (j = 0, j < gridY, j++)
        // EMPTY, SNAKE and APPLE are all global integers that refer to the
        // three possible square types in the grid.
        if (grid[i][j] != EMPTY)
          if (grid[i][j] == SNAKE)
            // Change the colour of the snake squares to dark grey.
            fill(64)
          else if (grid[i][j] == APPLE)
            // Change the colour of the apple squares to green.
            fill(0, 255, 0)
          endif

          // Draw this apple or snake square in the correct location.
          draw a rectangle with top-left corner (i * squareWidth, j * squareHeight) and
length squareWidth and height squareHeight
        endif
      endfor
    endfor

    // Draw the gridded lines that make up the grid.
    for (i = 0, i < gridX + 1, i++)
      draw a line from (i * squareWidth, 0) to (i * squareWidth, squareHeight * gridY)
    endfor
    for (i = 0, i < gridY + 1, i++)
      draw a line from (0, i * squareHeight) to (squareWidth * gridX, i * squareHeight)
    endfor
endprocedure
```

Level.update():

This algorithm is responsible for updating the current state of the level. It is called every frame and updates the snake and checks if the snake has collided with an apple, a wall or itself.

Pseudocode for this code is below:

```
procedure update()
    // This method moves the snake's head and checks if it has hit any walls or its tail.
    snake.update()

    // If the snake has died (i.e. hit a wall or tail) then reset the game
    if (snake.dead)
      snake = new Snake()
      resetApple()
      score = 0
    else if (snake.pos == apple.pos)
      snake.extend()
      resetApple()
      score++
    else
      // This updates the snake so that it moves a square in its current direction.
      snake.move()
    endif

    // This updates the values in the grid array to reflect the changes made this frame.
    updateGrid()
endprocedure
```

Snake.update():

This moves the snake's head in the direction it is currently facing and then checks if it has hit any walls or parts of its tail.

Pseudocode for this algorithm is below:

```
procedure update()
    // pos is the vector representing the current position of the snakes head.
    // direction is the direction the snake is facing (updated by the user's input)
    pos.add(direction)

    if (body.contains(pos) OR pos lies outside the boundaries of the grid)
      dead = true;
    endif
endprocedure
```

Main class:

The 'Main' class is the primary class that contains the methods that will be executed once the program runs. Although not explicitly called 'Main' in the Processing IDE, it functions similarly to the 'Main' class in a Java program. Here the main functionality of the program will take place; determining what happens during initialisation of the program and each frame the program runs. In Processing, all variables and methods declared in this class are public and global.

Attributes:

- SNAKE – This is a global constant that represents a square that is taken up by the snake in a grid. It is not stored in the level class as in further prototypes, multiple classes will use these variables and multiple level classes will be created. Variables declared in Main in the Processing language are global.
- EMPTY – This is a global constant that represents an empty square in a grid.
- APPLE – This is a global constant that represents a square that is taken up by an apple in a grid.

Prototype One:

- snakeWidth – This determines the width of the space the Snake grid takes up in pixels without affecting the dimensions of the grid.
- snakeHeight – This determines the height of the space the Snake grid takes up in pixels without affecting the dimensions of the grid.
- level – This is an instance of the class 'Level'. This variable holds all the details about the Snake and its grid, as well as a running score.
- gridX – This is a global definition of the width of the level's grid array. This is made global as once there are multiple Level instances in future prototypes, they will all share the same dimensions.
- gridY – This is a global definition of the width of the level's grid array.

Methods:

- setup() – This is a method defined by the Processing language. Any code in here is executed once at runtime and not repeated. All initial code to be executed is located here.
- draw() – This is a method defined by Processing. Code in here is executed every frame; similar to a while(true) loop. All code for updating the positions of the Snake and checking for collision will be located here, along with all functions for drawing the Snake grid on the screen.
- keyPressed() – This is a method defined by Processing. Code in here is executed whenever a key on the keyboard is pressed. I can use variables inside this function to access the key pressed. This will be used for changing the snake's direction.

Level class:

'Level' is the class that holds the grid array that the snake moves around, as well as the snake and the apple. Multiple instances of this class will be created in future prototypes to allow more than one snake to play at the same time.

Attributes:

- grid – This is a private 2D integer array of values that correspond to a blank square, square with a snake inside and square with an apple. This is the grid that will hold the current state of the game and will be visualised to the user.
- snake – This is a public instance of the Snake class that implements a snake that can navigate through the grid. Properties such as its current position will be accessed.
- apple – This is a private PVector attribute. PVector is a data type implemented by Processing used to store and manipulate vectors. I am only concerned about the apple's location, so vector coordinates are all that is needed.
- score – This is a public int that stores the running score of the game. It will increment whenever an apple is eaten.

Methods:

- Level() – This is the constructor for the Level class. This constructor initialises the grid, snake and apple.
- show() – This is a public void that displays the current state of the grid so that the user can visualise it.
- update() – Update is a public void. Whenever this is executed, the current position of the snake is updated based on the direction it is facing and checks are made to see if the snake is still alive or not. It will also check if the snake has eaten the apple and will update the snake accordingly.
- updateGrid() – This private method takes all of the location data for the snake and apple to create a new grid to display to the user. It discards the previous grid and creates a new one after the snake has been updated.
- resetGrid() – This private method is a shorthand way of initialising the grid to all empty values.

- set(PVector pos, objectType int) – This private method implements an easier way to update a square in the level's grid. The position contains the location to be updated and the objectType represents the value that the square is updated to. objectType will be one of the three global constants defined in the Main class.
- resetApple() – The private method resetApple finds a location for the next apple to appear that is not taken up by the snake's body.

Snake class:

The 'Snake' class holds all parts of the snake's body, current direction, and its current state. Its methods allow the snake to move and grow.

Attributes:

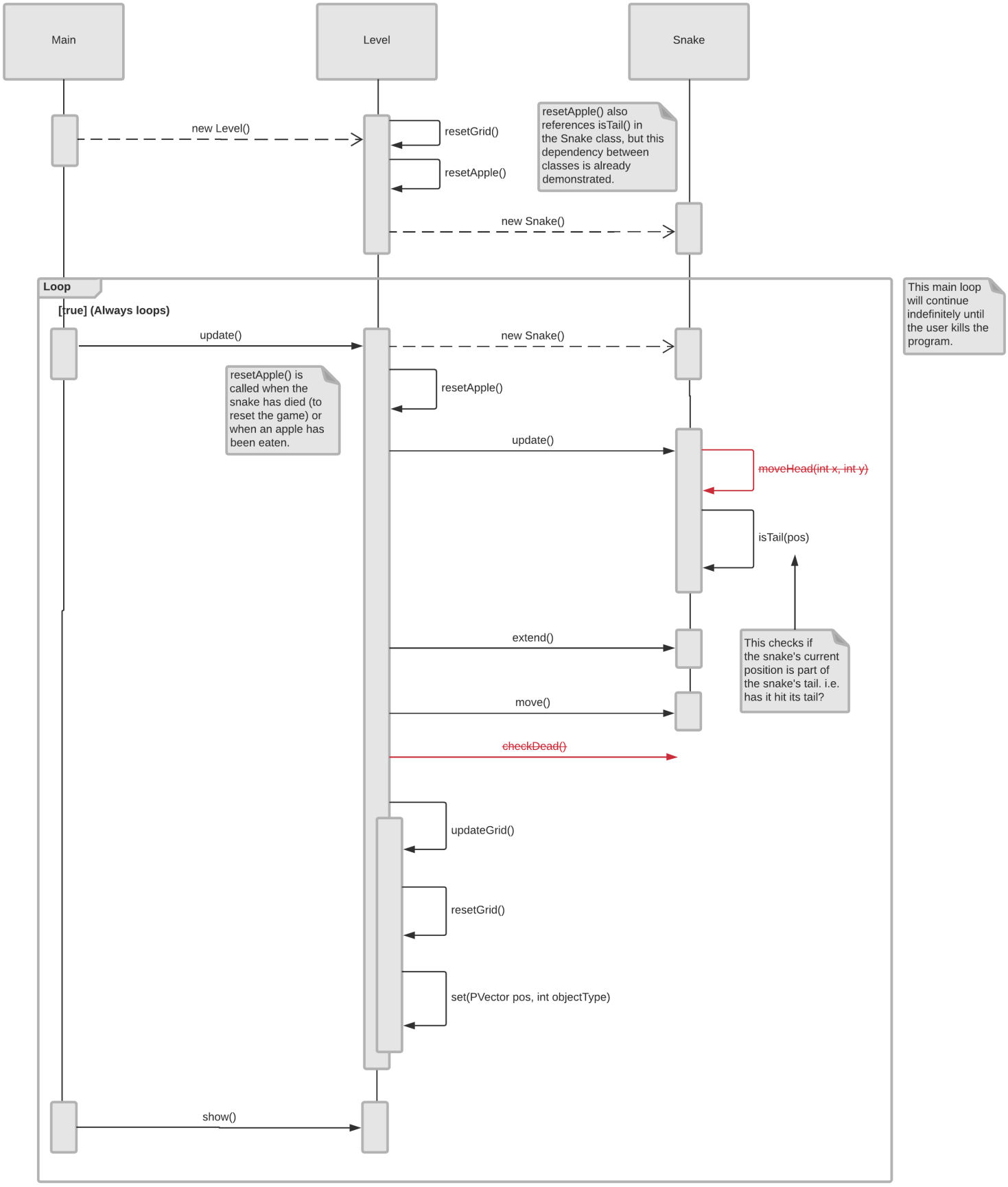
- pos – This is a public PVector which stores the current position of the head of the snake. This is updated whenever the snake moves.
- body – This is a private PVector list that stores all the locations of the parts of the snake. This is required when the snake gets longer as its tail curves when the head changes direction.
- direction – This public PVector stores the vector that will next be added to the current position of the snake to move it in a specific direction. It will be updated whenever an arrow key is pressed.
- dead – This is a public boolean to signify if the snake is dead or not. It will be used when the game needs to reset or to tell the program that its fitness needs to be calculated in later prototypes.

Methods:

- Snake() – This is the constructor for the Snake class. The snake is initialised with a length of 1 meaning that the body list has a length of 1 to begin with. Its initial direction is a vector pointing right.
- update() – This method updates the position of the snake and then checks if the new position is part of the snake's body or outside the boundaries of the grid, and therefore dead or not.
- extend() – Extend is called when the snake eats an apple. This adds an element to the body list so that the snake grows in length.
- isTail(PVector vector) – This public function will return true if the vector parameter is part of the snake's list of body vectors or false if it is not. The Snake's update method will use this function when checking if the snake is dead.

Prototype One:

Sequence Diagram:



James Ball

Prototype One:

This diagram shows the dependencies between classes and the methods they use. Methods that are red and crossed-out were removed from the prototype at some stage in the development log below. The section of the diagram before the loop is the initialisation of the program. The level class is initialised, and all variables are declared in their initial state. Once in the loop, the Snake game begins. The snake moves in the direction specified by its 'direction' attribute and all checks are performed to see if the snake has eaten an apple or died.

After the snake has moved or eaten an apple etc. the grid is updated and then displayed to the user. If the user presses an arrow key, the direction of the snake will be changed appropriately and it will move in that direction in the next level.update() phase.

Development Log:

Log 1 - November 1st:

Issue:

I realised that gridX and gridY make more sense as global variables than as attributes to the Level class as multiple classes will access these variables and all instances will share the same values.

Resolution:

I have moved these attributes and removed parameters from constructors that required these values. These constructors can now access gridX and Y directly, so they don't need an x and y value given to them.

Log 2 – November 1st:

Issue:

The direction variable in the Snake class would be better suited as a PVector rather than an int as PVectors can explicitly define the direction the snake moves in, rather than a later conversion to a PVector.

Resolution:

I have changed direction to a PVector, rather than an int. These changes have been reflected by removing the moveHead void, that was previously required to move the snake's head in a direction. This can now be easily completed in the snake's update void.

Log 3 – November 1st:

Issue:

The hitTail() void in the Snake class should be generalised to allow me to test if any PVector is part of the snake - not just the snake's head. I need this functionality in the resetApple() method, so generalisation is required.

Resolution:

I have renamed hitTail to isTail and added a parameter 'vector' which will be checked against every element in the 'body' list. It will return true if the vector is part of 'body'.

Log 4 – November 2nd:

Issue:

The checkDead void in the Level class should be moved to the Snake class as all the variables it uses are global or specific to the Snake class.

Resolution:

I have moved the method to the Snake class.

Log 5 – November 2nd:

Issue:

The checkDead void should be completely removed and this code should instead be located inside the Snake class update method. Snake.update() and Snake.checkDead() are always executed together, so it makes sense for them to share a method.

Resolution:

I have moved the code in the checkDead void into the Snake's update function.

Log 6 – November 2nd:

Issue:

I had a NullReferenceException from Java relating to my resetGrid method.

Resolution:

I realised that I hadn't initialised the array before I assigned values to elements within it. After re-running, it works as expected.

Log 7 – November 7th:

Issue:

My program crashed with a NullPointerException in the set() method.

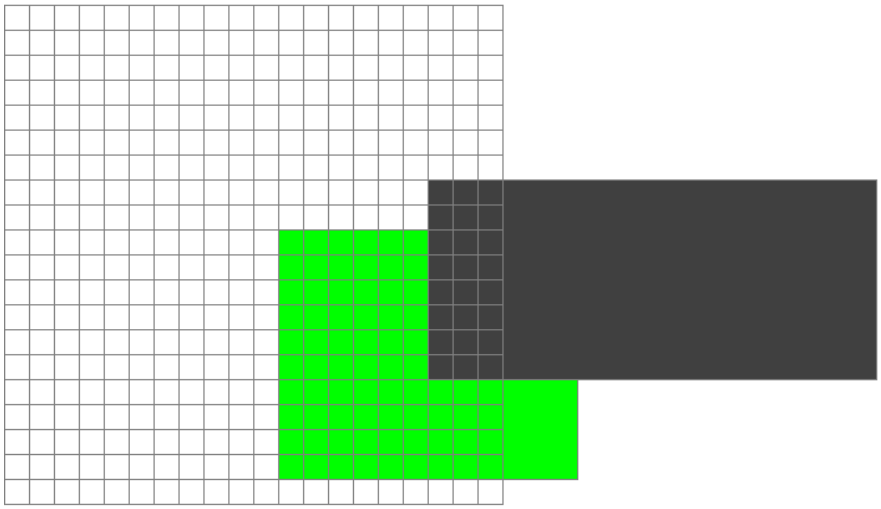
Resolution:

I debugged the program by looking at updateGrid(), the only place that uses set(). Using a breakpoint in updateGrid(), I stepped through and saw there was an issue when I set the location of the apple. I then saw that apple was null. This means that apple was never initialised, meaning the while loop in resetApple() was not working. I finally realised that apple should be given a random value before the while loop, as well as inside the while loop - otherwise it would always have a null value. After rerunning, the bug was solved.

Log 8 – November 8th:

Issue:

There is a problem with the way squares are rendered on the grid. Squares are much larger than they should be. An image explaining this is below:



Resolution:

This is clearly an issue with the show function in some way. Upon inspection, I realised that the rect() function in Processing defines the starting x, y coordinate and then the width and height. I thought it was the first x, y coordinate and the second x, y coordinate, causing this graphical bug. After changing this, it ran exactly how it should.

Log 9 – November 12th:

Issue:

After adding keyboard control of the snake, I realised the snake couldn't see that it touched an apple.

Resolution:

I thought it may be related to how I was checking if two PVectors had the same value as I was comparing them directly, rather than their respective x and y values. I changed all direct

comparisons with (vector.x == otherVector.x && vector.y == otherVector.y) so they would be correctly compared. After changing this, the snake could collide with apples.

Log 10 – November 12th:

Issue:

After fixing log 9, when the snake collides with an apple, it dies.

Resolution:

I suspected this had something to do with the isTail function, but after a lot of debugging and research, I realised that when the snake moves or extends, it is adding the variable Snake.pos to Snake.body, which means it is being added by reference, rather than value, so any further modifications to Snake.pos will be reflected in the variable previously added to the Snake.body list. I have now changed the code so in extend() and move(), new PVectors are created with the same values as pos. This effectively adds Snake.pos to Snake.body by value rather than by reference. After changing this code, the snake moves and extends as it should.

Testing

The Snake game involves no data processing, so there is no test data to test this program. Instead, user testing is more appropriate as it is a game and the user can define whether the game runs correctly or not. This testing section tests against each success criteria laid out in Analysis.

These are the results of testing each success criteria:

Success Criteria	Test Description	Expected Outcome	Actual Outcome
1	I need to test the collision system for the snake. The game should reset if you hit your tail or the wall.	The snake should not have any issues going around the edge of the grid, but if they hit their tail or hit the wall, the game should reset.	After testing with a user, the game performed as expected. This functionality can be evidenced in the video labelled Prototype1.mp4 within the Prototype1 folder. A survey with user feedback can be seen under 'User test survey' below.
2	The extend method on the snake should be tested to ensure that it is called whenever the snake eats an apple and to ensure it functions as expected.	The snake should grow in length whenever an apple is touched.	Multiple games were played in which multiple apples were collected and the length of the snake grew as expected. Growing in length is evidenced in the Prototype1.mp4 video. See 'User test survey' for user feedback on this functionality.
2	I need to test the movement of the snake, making sure it can turn and retain the previous segments of the snake after it changes direction.	The snake should change direction in a way that means all body segments before the turn remain in the same location before they exceed the length of the snake. This pattern of movement can more easily be explained by looking at the initial sketch.	After playing continually with a user, the snake moved as expected with a snake game. This can be shown in the screenshot under the title 'Snake movement' and in the Prototype1.mp4 video. See 'User test survey' for user feedback on this functionality.
3	I need to confirm that my program can be easily adapted to display multiple snakes at once in further prototypes.	Minimal changes in my code should allow more than one snake to be rendered and controlled at a time.	I have coded the program in a way that allows new instances of snakes and levels to be created so that more than one snake can be rendered at the same time. A screenshot demonstrating the initialisation of this can be seen below, labelled 'Initialisation of a new Level'.

4	I need to make sure that the snakes can be easily controlled by a neural network in later prototypes.	I should be able to change the inputs for the snake easily in the code to allow inputs from a neural network.	To change the inputs for the snake, I only need to change the function that changes the direction of the snake. Each snake can be given its own neural network and the output of the neural network can determine what direction it moves in next. I have demonstrated how this could be implemented below under the heading 'Changing snake inputs'.
5	I need to ensure the grid's size can be easily changed.	I should be able to easily change the size of the grid in my program and snake should still be playable at this size.	After changing the value of gridX and gridY to 30 I ran the game again and performed user testing over multiple games to look for any issues and I noticed that grid lines were extending out the sides of the grid. I fixed this issue and re-ran the program and it worked as expected as evidenced in the 'Changing grid size' section. Other than this, it worked as expected and was completely playable.
6	I need to test that frame-based movement works as expected.	I should be able to specify the framerate of the program and the snakes should move according to this, rather than moving every x seconds at a high framerate.	The program has been written so that the snake will update every frame, so I can specify the framerate to change the speed of the game. In further prototypes the framerate will be as high as possible to ensure maximum performance.

User test survey:

The user was asked to play for 5 minutes, first testing the collision system by crashing into walls, then growing and testing the turning and general movement of the snake, and then crashing into the snake's own tail to test the collision there as well. Below is the feedback from the user on each part they were asked to test:

Collision system (Crashing into tail and walls):

The user started the test by repeatedly crashing into walls. All 4 walls were tested, and the boundaries of these walls were also checked to see if they matched the visual representation correctly. All walls caused the snake game to reset, but the snake can still move along the edges of the wall, so both the walls and wall boundaries work as expected. Similarly, the user then crashed into the snake's own tail after increasing the length and direction of the snake. This was tested in various scenarios, such as the snake going back on itself, and hitting itself from different sides. All results were as intended, but the user mentioned an issue they had with the game, which made it different to other Snake games they had played. This is detailed in the 'Other comments' section.

Increasing the length of the snake:

The user was told to collect apples and try to reach a high score. The collision of apples and the snake's head was tested and the extend() function in the Snake class was being tested. After the user had tested this behaviour, I could verify that the game functions as expected as the user had no issues with collision or movement when the snake was long.

Changing the direction of the snake (When grown and small):

I told the user to change directions, both quickly and slowly to see how the game would react. If the user input two movements in quick succession, only the last movement would register. This could be an issue for a Snake game being played by a human, but as the game will be adapted to an AI agent

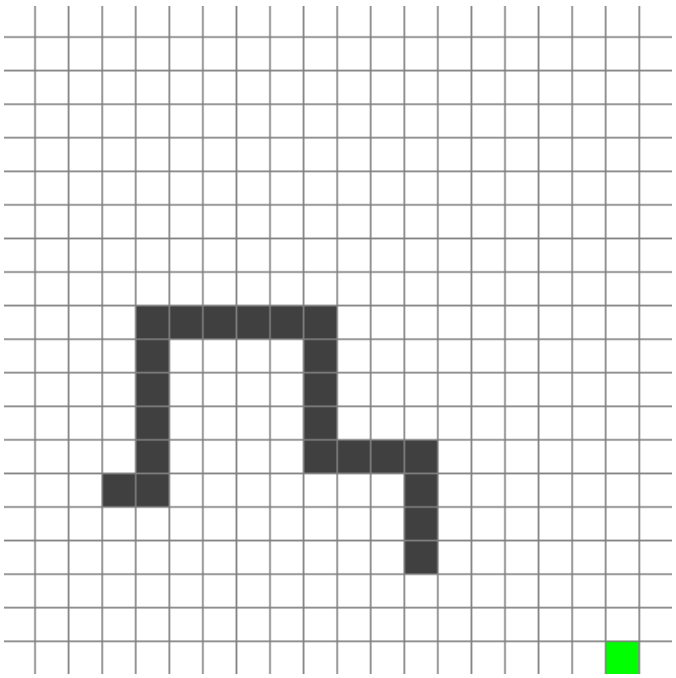
that is restricted to one input per frame, this will not be an issue in further prototypes. When the snake is small, there is no tail to trail behind so movement is trivial. At increased lengths, the tail of the snake should trail behind it, which it did in this user test. The user once again had no issues.

Other comments:

The main issue this user had with this prototype was the fact that if you move in the opposite direction to which you are currently facing, you will die straight away as the game registers that you have hit your own tail. He stated he would prefer if this movement was prohibited. Although I understand this for a human playthrough of Snake, this movement should not occur for an AI agent, as they will quickly learn that moving wherever a snake is will kill them. This means they will be heavily punished if they turn back on themselves.

Snake movement:

You can see in this image how the snake has grown and changed direction, like the original Snake game. The grey pixels are the snake and the green pixel is the apple.

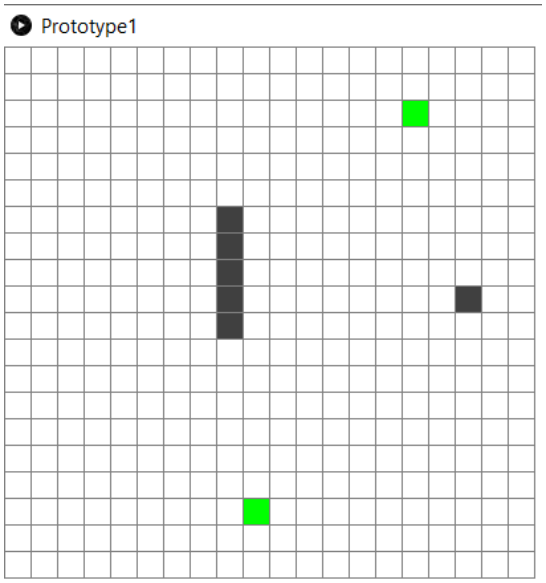


Prototype One:

Initialisation of a new Level:

The original code has been modified so that another level is initialised called level2. This can then be updated and shown along with the first level so that two can be displayed at the same time, as shown in the screenshot on the right. If I want to render hundreds of snakes at once, this can be easily achieved by creating a Level array and rendering all levels in this array. You can see that two snakes are rendered (grey pixels) and two apples are rendered (green pixels).

```
10 int gridX = 20;
11 int gridY = 20;
12 Level level = new Level();
13 Level level2 = new Level();
14
15 /* The setup method is initial
16    program execution. */
17 void setup() {
18   /* The following defines the
19   size(1000, 1000);
20   frameRate(10);
21 }
22
23 /* The draw method is executed
24 void draw() {
25   background(255);
26   level.update();
27   level.show();
28   level2.update();
29   level2.show();
30 }
```



Changing snake inputs:

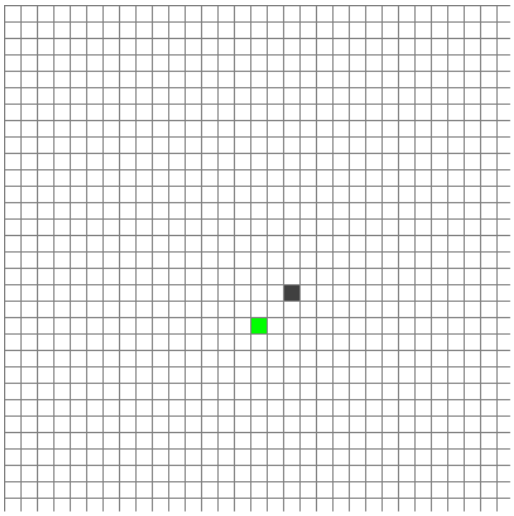
Instead of changing the direction of the snake in the main class, I would manage their movements within the snake's update function based on the output of the neural network. The output will determine which direction the snake moves in next. This is easy to do; hence this test is successful.

```
/* keyPressed is executed whenever a key is pressed. keyCode stor
   change the direction of the snake when the arrow keys are pres
void keyPressed() {
  switch(keyCode) {
    case UP:
      level.snake.direction = new PVector(0, -1);
      break;
    case RIGHT:
      level.snake.direction = new PVector(1, 0);
      break;
    case DOWN:
      level.snake.direction = new PVector(0, 1);
      break;
    case LEFT:
      level.snake.direction = new PVector(-1, 0);
      break;
  }
}
```

Prototype One:

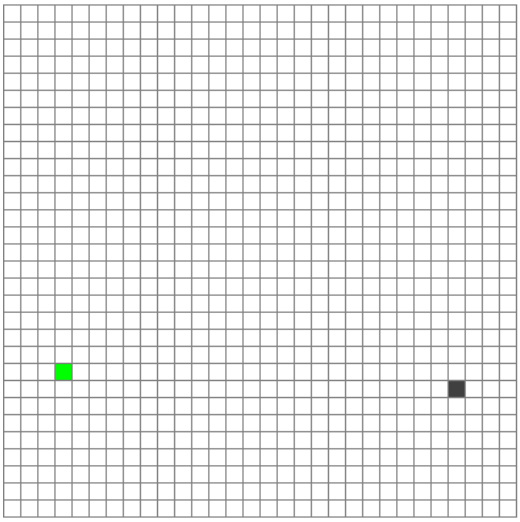
Changing grid size:

Before I fixed the issue, the grid displayed as below. The grid extends past the point it should on the right and on the bottom. I have also included the code before the bug was fixed:



```
  stroke(128);
  /* These two for loops draw the gridded area. */
  for (int i = 0; i < gridX + 1; i++) {
    line(i * squareWidth, 0, i * squareWidth, snakeHeight);
  }
  for (int i = 0; i < gridY + 1; i++) {
    line(0, i * squareHeight, snakeWidth, i * squareHeight);
  }
}
```

Below is the grid after I fixed the bug. I only changed a small part of the code as you can see below. snakeWidth and snakeHeight have been replaced.



```
  stroke(128);
  /* These two for loops draw the gridded area. */
  for (int i = 0; i < gridX + 1; i++) {
    line(i * squareWidth, 0, i * squareWidth, squareHeight * gridY);
  }
  for (int i = 0; i < gridY + 1; i++) {
    line(0, i * squareHeight, squareWidth * gridX, i * squareHeight);
  }
}
```

Expected and overall outcome of user testing:

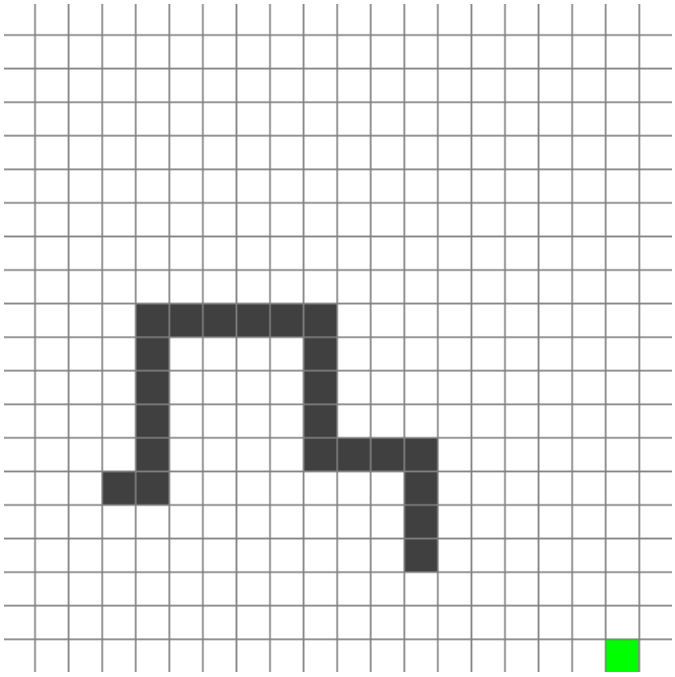
The expected outcome for this user test is for users to be able to play a game of Snake that is mechanically identical to the original.

My focus group played the game and they though it played exactly like a Snake game but identified one issue – when the snake moves back on itself it dies. This is because it 'hits its own tail' when it is moving in the opposite direction.

Although this is an issue for a Snake game played by people, the AI will learn not to move backwards as this will instantly kill the snake. This means that this won't be an issue in further prototypes that are not controlled by people.

Evidence of this prototype being successful is provided as both a screenshot and in video form. The video is in the Prototype1 folder and the screenshot can be seen below:

Prototype One:



In this photo, you can see how the snake has grown in length and can bend and change direction. I changed the size of the grids for each user and had them test the game on different size grids. Grids that were not an X by X size (i.e. 20x20) rendered but were stretched. Although this is a visual bug that could be fixed, all grid sizes in later prototypes will be square, so it is an unnecessary fix.

Evaluation:

Success criteria:

Meeting each success criteria for prototype 1 has been heavily evidenced in the testing section above. I have performed a test against each success criteria, including subjective criteria that is code-related, such as success criteria 3, 4 and 6. To test the more subjective criteria mentioned, I briefly changed parts of the original code to demonstrate how easily the code can be adapted to being controlled by a neural network, or to have an uncapped framerate, for example.

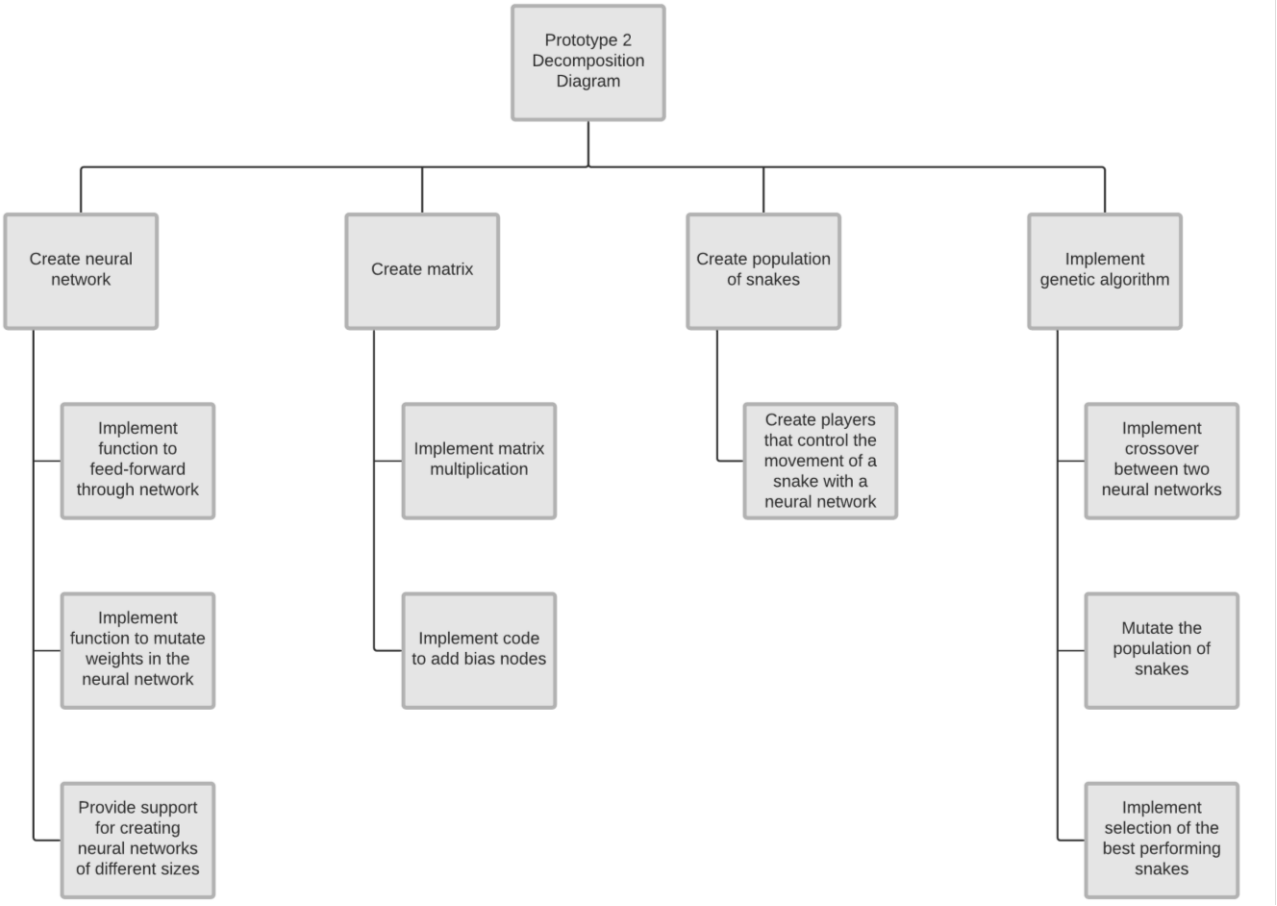
As demonstrated by this testing, all criteria have been met, making this program ready for prototype 2.

The ‘User test survey’ section demonstrates that my focus group have not raised any major issues with the first prototype. Therefore, there are no problems moving forward into the second prototype that need to be fixed before development on the next prototype starts. The issues that were raised were relevant to a human-played Snake game (such as multiple inputs in the same frame not registering), but moving forward, these will not affect development as further prototypes remove user input as it is AI-based.

Prototype Two:

The second prototype aims to implement the genetic algorithm along with the neural networks, matrix multiplication, and other code required to implement the AI and allow the AI to learn. This prototype is the most in-depth due to the complexity of many of the algorithms implemented. One of the main things I must consider during development of this prototype are the hyper-parameters (i.e. the dimensions I should use for the neural network, mutation rate, crossover algorithm etc.) as these must be adjusted greatly to reduce the time spent training the AI and to maximise the peak performance of the AI. Hyper-parameters are further explained in the Neural Network research section in Analysis.

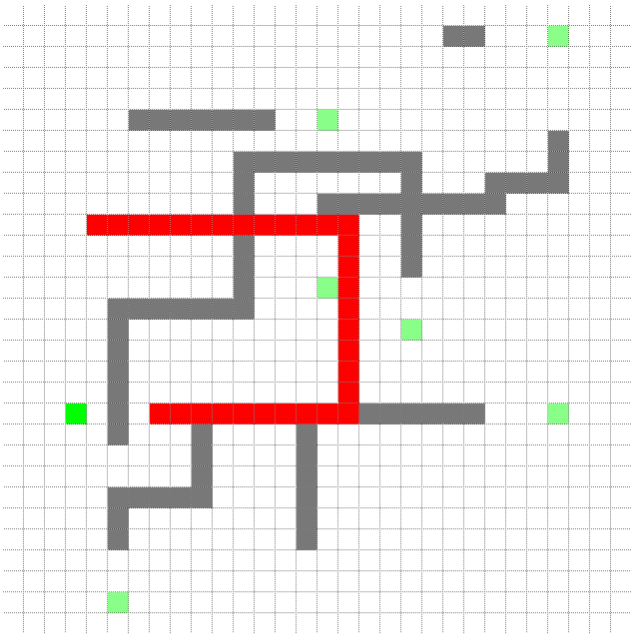
Decomposition Diagram:



This shows the sub-problems I am decomposing prototype 2 into. The four main sub-problems for this prototype are to implement classes to create neural networks, create matrix classes, create a population of snakes and implement the genetic algorithm. This prototype is the most complicated of the three, so there are many sections within these sub-problems.

Prototype Two:

Initial Sketch:



Players rendered: 7
Number dead: 350
Framerate: 235
Score of best: 23

The second prototype is all about implementing the algorithms related to the genetic algorithm to reach a stage in which the AI can learn how to play Snake overtime. In this sketch, 7 snakes are shown on the screen at once, which is an abstraction of how many snakes will be playing the game at any one time. There will be hundreds of snakes playing, but only a few being displayed.

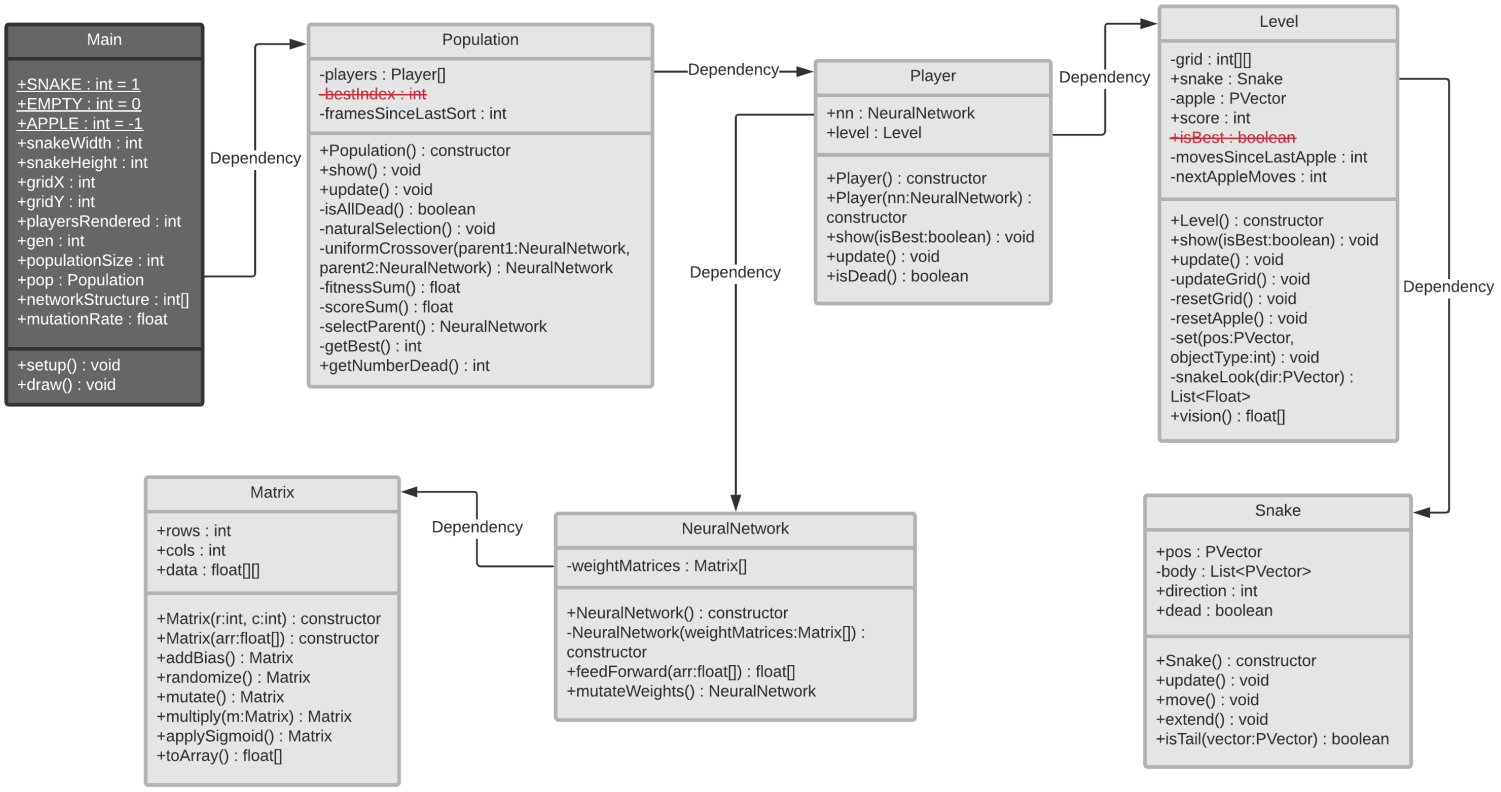
The red snake is the snake with the highest score at that moment in time. This is highlighted so that the user can visually see the current state of the AI and see how well it is performing. This snake's apple is also a brighter green to distinguish it from other apples.

There are some statistics for debugging purposes below the playing grid. It shows the current number of snakes displayed, the number of snakes that have already died, the current framerate and the score of the best snake, which is the snake highlighted in red.

Prototype Two:

Classes:

Class diagram:



Explanation of key algorithms:

Below is pseudocode for the fundamental algorithms of prototype 2. This includes all major algorithms for the genetic algorithm and neural network.

Level.snakeLook():

This acts as the input for the snake's neural network or 'brain'. It takes a 2D vector as a parameter and returns a float array of the reciprocals of the distances (i.e. 1/distance) from the snake's head to the snake's own body, the apple and the wall of the grid respectively. If the snake's body or apple are not in this direction, 0 is instead returned. Pseudocode below:

```
private function snakeLook(Vector direction)
    // copy snake.pos so we can modify it without changing the original.
    Vector snakePos = copy(snake.pos)
    float[] vision = new float[3]
    int distance = 1

    for (int i = 0, i < vision.length, i++)
        vision[i] = 0
    end for

    // adds the direction parameter to the snake's position and checks to see if it still
    // lies in the grid boundaries.
    while (withinBounds(snakePos.add(direction)))
        // checks if the square at snakePos is part of the snake's body and makes sure this
        // vision element hasn't already been assigned.
        if (grid[snakePos.x][snakePos.y] == SNAKE && vision[0] == 0)
           vision[0] = 1/distance
        end if

        if (grid[snakePos.x][snakePos.y] == APPLE && vision[1] == 0)
```

```
            vision[1] = 1/distance
        end if

        distance++
    end while

    // the distance to the wall is the distance variable once snakePos is no longer within
    // the boundaries of the grid.
    vision[2] = 1/distance

    return vision.toList()
end function
```

This function is used in Level.vision() to build together all 24 inputs for the snake – corresponding to three inputs in all eight directions.

Level.update():

This updates the state of a specific level in the population. The pseudocode below shows how the snake is killed if the snake takes too long to eat an apple and shows what happens when the snake eats an apple.

```
public procedure update()
    snake.update()

    // if the snake runs out of moves for this apple...
    // nextAppleMoves stores the number of moves allowed to get to the next apple.
    // this stops the snake getting stuck in loops.
    if (movesSinceLastApple > nextAppleMoves)
        snake.dead = true
    end if

    if (!snake.dead)
        // if the snake's head is on the apple...
        if (snake.pos == apple)
            snake.extend()
            resetApple()
            score++
            // this is explained in development log 4.
            nextAppleMoves = 200 * logBase3(snake.body.length) + 300
        else
            snake.move()
        end if

        movesSinceLastApple++
        updateGrid()
    end if
end procedure
```

Matrix.multiply():

This function takes a matrix as a parameter and multiplies it with the Matrix object it is called from. The process of matrix multiplication is explained in the Analysis section in Research of Algorithmic Methods. The rows and cols attributes refer to the number of rows and columns of that specific matrix. The pseudocode is below:

```
public function multiply(Matrix m)
    // initialises the resultant matrix with the correct dimensions.
    Matrix r = new Matrix(rows, m.cols)

    // for matrix multiplication to be valid, this matrix's number of columns must be equal
    // to the number of rows in the parameter matrix.
    if (cols = m.rows)
        // this is the multiplication process.
        for (int i = 0, i < rows, i++)
```

```
            for (int j = 0, j < m.cols, j++)
                float sum = 0
                // this calculates the element at row i and column j in Matrix r.
                for (int k = 0, k < cols, k++)
                   sum += data[i][k] * m.data[i][k]
                end for

                r.data[i][j] = sum
            end for
        end for
    else
        // throw an exception if the dimensions are not correct.
        throw exception
    end if
end function
```

NeuralNetwork.feedForward():

This function takes an array as an input and feeds this input through the neural network. This means the input passes through each layer in the network and is multiplied by the networks weight matrices and an activation function is applied at each layer. The outputs produced by this NN are returned. The pseudocode below shows how this is performed:

```
public function feedForward(float[] arr)
    // checks the input array matches the number of neurons in the input layer of the NN.
    if (arr.length == networkStructure[0])
        // creates the layer we are currently operating on.
        Matrix currentLayer = new Matrix(arr)

        // progresses through the NN by multiplying the respective weight matrix by the
        // current layer with an added bias node. The sigmoid activation function is then
        // applied.
        for (int i = 0, i < weightMatrices.length, i++)
           currentLayer = weightMatrices[i].multiply(currentLayer.addBias()).applySigmoid()
        end for

        return currentLayer.toArray()
    else
        // throw an exception if the input array is the wrong length.
        throw exception
    end if
end function
```

Population.show():

This procedure sorts the players array in descending order by the score of each player every 200 frames and then shows the number of players dictated by the playersRendered attribute in the Main class. The pseudocode is below:

```
public procedure show() {
    if (playersRendered > 0)
        // if it is time for another sort...
        if (framesSinceLastSort > 200)
           sort(players, descending)

           framesSinceLastSort = 0
        end if

        for (int i = 1, i < playersRendered, i++)
           players[i].show(false)
        end for

        // show(true) means this snake will be highlighted as the best performing snake.
        players[0].show(true)
    end for
```

```
    framesSinceLastSort++
    end if
end procedure
```

Population.naturalSelection():

This method performs the genetic algorithm and is executed at the end of a generation. The genetic algorithm is explained in detail in the Analysis section. Pseudocode below:

```
private procedure naturalSelection()
    Player[] nextGen = new Player[populationSize]

    // the best player from the last generation moves the new generation to ensure the next
    // generation's best shouldn't perform worse.
    nextGen[0] = new Player(players[getBest()].nn)

    for (int i = 1, i < populationSize, i++)
        // create each player in the new generation by performing selection, crossover and
        // mutation.
        nextGen[i] =
        new Player(uniformCrossover(selectParent(),selectParent()).mutateWeights())
    end for

    gen++

    players = nextGen
end procedure
```

Population.uniformCrossover():

This function performs uniform crossover on two neural networks. This means that a child NN is created using the two parent's weight matrices. There is a random but equal chance that either parent NN's weights are used for a specific weight in the child's weight matrix. This means that roughly half of the child's weights are from parent1 and the rest from parent2. Pseudocode below:

```
private function uniformCrossover(NeuralNetwork parent1, NeuralNetwork parent2)
    NeuralNetwork child = new NeuralNetwork()

    // loop through every weight in all weight matrices.
    for(int i = 0, i < parent1.weightMatrices.length, i++)
        for (int j = 0, j < parent1.weightMatrices[i].rows, j++)
           for (int k = 0, k < parent1.weightMatrices[i].cols, k++)
                // there is a 50% chance the child inherits this weight from a specific
                parent.
                if (random(1) > 0.5)
                   // set the weight in the child's weight matrix to the parent's weight.
                   child.weightMatrices[i].data[j][k] = parent1.weightMatrices[i].data[j][k]
                else
                   child.weightMatrices[i].data[j][k] = parent2.weightMatrices[i].data[j][k]
                end if
           end for
        end for
    end for

    return child
end function
```

Population.selectParent():

This function returns a random player from the players array. The higher the score of the player, the more likely it is to be chosen. I square the score of the players in this calculation to artificially inflate the scores of high-scoring snakes to make it even more likely they are chosen. Pseudocode below:

```
private function selectParent()
    // generates a random cut-off point between 0 and the fitness sum.
    float threshold = random(fitnessSum())
    float total = 0
```

```
    for (int i = 0, i < players.length, i++)
        // raises the player's score to the second power.
        total += pow(players[i].level.score, 2)

        // if the player just added passes the random threshold, return this parent's NN.
        if (total > threshold)
           return players[i].nn
        end if
    end for

    return null
end function
```

Main:

The Main class has the same function as detailed in Prototype 1, but Prototype 2 adds some functionality which is detailed in this section. Some added attributes are hyper-parameters for the neuroevolutionary process (i.e. mutationRate, populationSize).

Attributes:

- playersRendered – This integer specifies the number of snakes that are currently rendered on the screen at any one time. This will be able to be controlled by the user and is necessary to increase the framerate as the program runs as it will reduce the number of snakes on the screen.
- gen – This is the current generation of the population of snakes. After every snake has died, this variable is incremented.
- populationSize – This is the size of the population of snakes in each generation. This is a hyper-parameter that will be experimented with later in this prototype.
- pop – This is the Population object that holds all Player objects and performs the genetic algorithm.
- networkStructure – This is an array of integers representing the dimensions of the neural network that all the players will use. The first and last values represent how many neurons are in the input and output layers and any other values in the array are the size of any hidden layers. This allows me to easily change the structure so I can test the most efficient structure.
- mutationRate – This is another hyper-parameter that determines how likely a specific attribute is to be mutated during the mutation stage of the genetic algorithm. It should lie within the range 0-1, corresponding to the likelihood of mutation.

Methods:

- setup() – Along with the functionality detailed in Prototype 1, the setup procedure now initialises the pop attribute in this class.
- draw() – Unlike Prototype 1, the draw procedure no longer updates snake-related movement. It is now responsible for updating the population of snakes, drawing them and checking when they are all dead so that the next generation can be generated. This procedure is also responsible for drawing information about the population to the screen (e.g. the number of players rendered) as shown in the initial sketch.
- keyPressed() – Unlike Prototype 1, this procedure does not control snake movement, as this is now purely controlled by the neural network. It is now responsible for seeing when the user has pressed the plus or minus keys to increment or decrement the playersRendered attribute respectively.

Prototype Two:

Level:

This class is fundamentally the same as in Prototype 1, but now introduces a few attributes and methods to make it applicable for the genetic algorithm and AI. Level.show() still has the same function, but now includes a parameter for when a snake is to be rendered as the current best-performer.

Attributes:

- isBest – This is a boolean that is true if this level contains the snake that is currently performing the best. It remains false otherwise. This allows the snake to be drawn differently if it is the best performing snake. (This was later removed in development log 11).
- movesSinceLastApple – This integer will increment every move that the snake in this level takes and is reset to 0 when the snake eats an apple. Once it exceeds a pre-defined value, it will be assumed that the snake is stuck, and it will then be declared dead so that no snake is stuck in an infinite loop.
- nextAppleMoves – This private integer stores the number of moves given to the snake to get to the next apple. If they do not get to the apple in this number of moves, they are killed.

Methods:

- snakeLook(PVector dir) – This returns a list of three floating-point values. Two of the values are the reciprocal of the distance (i.e. 1/distance) from the snake's head to the wall or to part of the snake's body in a given direction. The other value is simply 1.0 if the apple is in that direction, or 0 otherwise. E.g. if the wall is very far away, the value returned will approach 0 and if it is very close, the value will approach 1. This direction is provided as a parameter (i.e. PVector dir). If there is not a body-part or apple in a given direction, 0 will be returned.
- vision() – This returns an array of 24 floating-point values that correspond to the 3 values returned by using snakeLook() in all eight directions (i.e. up, down, left, right and all diagonals) around the snake's head. This makes up the input for the neural network and is used by the Player class to feed-forward in the player's neural network.

Snake:

The Snake class is identical to the class in Prototype 1.

Population:

The Population class is responsible for managing all 'players' or snakes. It initialises all the levels, snakes and neural networks and performs the fundamental functions involved in the genetic algorithm, such as selection, mutation, crossover and creating new generations.

Attributes:

- players – This is a Player array which is the population of players that this class controls.
- bestIndex – This stores the index of the player and the respective snake that is currently performing the best in the current generation. (This has since been removed in development log 6).
- framesSinceLastSort – This private int stores the number of frames since the players array has been sorted, allowing for only the best snakes to be displayed.

Methods:

- Population() – This is the constructor for the population class. This initialises all players in the players array.
- show() – This is a public void that shows the number of snakes defined by the playersRendered attribute in the Main class. It will also show the snake that currently has the highest score.

- update() – This is a public void that updates all players every frame.
- isAllDead() – This is a private boolean that returns true if all player's snakes are dead. This is used in the Main class to trigger the next generation being generated.
- naturalSelection() – This private void is responsible for resetting all players once they have died and creating the next generation of neural networks/snakes. All selection, mutation and crossover processes occur here to generate a new generation of snakes that has learnt from the previous generation.
- uniformCrossover(NeuralNetwork parent1, NeuralNetwork parent 2) – This private function performs uniform crossover on two neural networks and returns the resultant NeuralNetwork. Uniform crossover involves choosing each weight for the child's NeuralNetwork from either parent1 or parent2. The parent chosen for each weight is completely random. This is explained further in the Genetic Algorithm section in Analysis.
- fitnessSum() – This private function calculates the fitness of each player and returns the sum.
- scoreSum() – This private function calculates the score of each player and returns the sum.
- selectParent() – This private function semi-randomly selects a NeuralNetwork from the players array so that the best performing players are more likely to be selected than poorly performing players. This is then returned to be used in the uniformCrossover function.
- getBest() – This private function returns the index of the snake that currently has the highest score.
- getNumberDead() – This public function returns the number of snakes in the player array that are currently dead.

Player:

The Player class holds the neural network for a snake along with a Level object to allow the Player class to control the snake's movement. It is responsible for processing the inputs and outputs of the neural network and then changing the snake's direction as a result.

Attributes:

- nn – This NeuralNetwork object is the 'brain' of the Snake AI for this player.
- Level – This is a Level object that holds information about the current state of this player's level, including the position of the apple and snake. It also manages the snake's movement and checks to see if the snake has died.

Methods:

- Player() – This is the Player class constructor. It creates a new NeuralNetwork object and Level object.
- Player(NeuralNetwork nn) – This is another constructor for the Player class. It creates a new Player using an already specified neural network which is used in the naturalSelection() method in the Population class.
- show(Boolean isBest) – This is a public void that is called every frame and simply renders the level.
- update() – This public void feeds-forward the snake's vision into the player's neural network and uses the output from the network to change the current direction of the snake. It executes every frame.
- isDead() – This is a public boolean that returns true if the snake has died and false if it is still alive.

NeuralNetwork:

This class is responsible for providing all the functions relating to NeuralNetworks, such as feeding-forward and comparing two neural nets.

Attributes:

- weightMatrices – This is an array of matrices that hold all the weights between layers in the neural network. There is a weight matrix between any two layers in the network, so there is one less weight matrix than the number of layers.

Methods:

- NeuralNetwork() – This constructor initialises all weight matrices randomly so all weights are a random decimal between -1 and 1.
- NeuralNetwork(Matrix[] weightMatrices) – This is a private constructor that is used for copying a neural network.
- feedForward(float[] arr) – This feed-forwards through the neural network when provided with a float array. Feeding-forward is explained in more detail on the Neural Network section in Analysis.
- mutateWeights() – This public function creates a copy of the neural network and then returns a mutated version.

Matrix:

The Matrix class is essentially a 2D array but with some helpful methods for dealing with neural networks, such as multiplication.

Attributes:

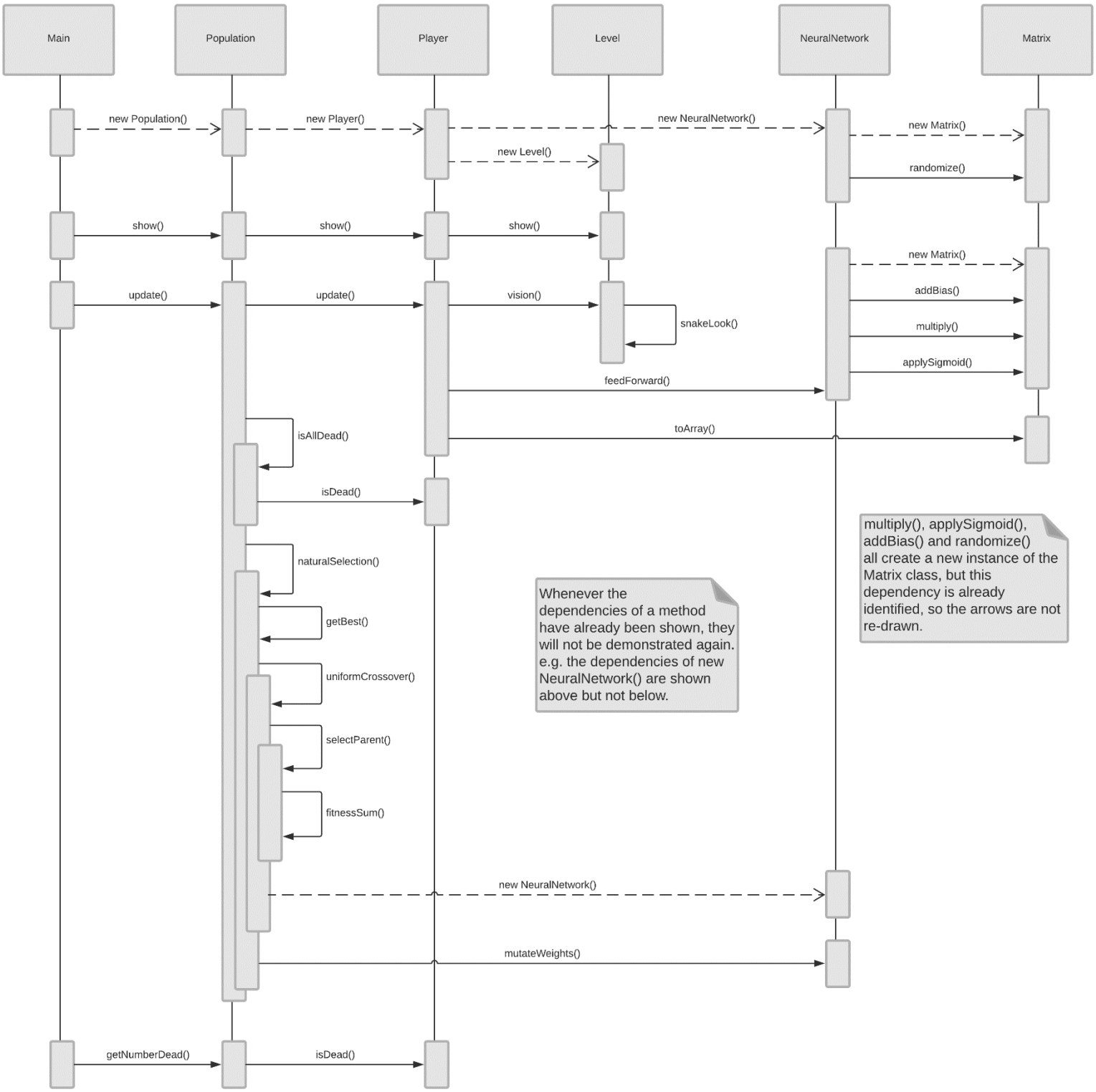
- rows – This specifies the number of rows in the matrix.
- cols – This specifies the number of columns in the matrix.
- data – This is the 2D float array that stores all the data for the Matrix class.

Methods:

- Matrix(int r, int c) – This is the constructor for the Matrix class. It creates a matrix with dimensions r by c and each value is set to 0.
- Matrix(float[] arr) – This constructor creates a matrix with one column consisting of the data in the arr array. This is useful when creating an input matrix when feeding-forward.
- addBias() – This public function returns a new Matrix with an extra bias node with a weight of 1. This bias node is explained in the Neural Network section in Analysis.
- randomize() – This public function returns a new Matrix with random values in the range -1, 1.
- mutate() – This public void iterates through each value in the matrix and there is a chance of that value being slightly changed.
- applySigmoid() – This public function returns a new Matrix that uses the values in the original matrix as an input into a sigmoid activation function. Sigmoid is the activation we will use for this program. Activation functions are mentioned in the Neural Network section in Analysis.
- toArray() – This public function converts the 2D array data into a 1D array by flattening the array. The resultant 1D array is then returned.

Prototype Two:

Sequence Diagram:



Dependencies already shown in the Prototype 1 sequence diagram have not been visualised again in this diagram, but any changes to the original diagram are present here. This diagram shows all dependencies introduced in Prototype 2. One change that is not shown is the addition of the Snake.outsideBounds() function, which the Level class depends on. This is not present as it was added later in development.

James Ball

Methods that are executed every frame are also displayed here through the update() method. This is important to consider as these methods are repeatedly executed and must be optimised.

Development Log:

Any errors I come across in development are tackled with in the Testing section, rather than in the development log as tests are performed during development. This log contains all changed functionality.

Log 1 – March 4th:

I am going to implement the Matrix, NeuralNetwork and Player classes before dealing with the Population class as this allows me to test the functionality of these classes instead of assuming they work. This helps development as the Population class assumes all aspects of the NeuralNetwork and Player classes work as expected.

Log 2 – March 6th:

I have decided that if a method is called on a Matrix object (e.g. matrix.randomize()), the original matrix data should be modified instead of creating a new Matrix object and returning this. This will reduce the amount of processing that occurs as the number of newly initialised Matrix objects is minimised. This optimisation cannot be made in methods such as addBias() that must create a new Matrix object as they change the dimensions of the original object.

I have also changed the mutate method to a function that returns a Matrix rather than mutating the matrix without returning it.

Log 3 – March 6th:

To test the Matrix class, I need an easy way to print the contents of the matrix to the console. After some research, I found a Java function Arrays.deepToString() which will format a 2D array as a string, allowing me to easily view the contents of the matrix. This also requires a reference to java.util.Arrays.

PREFACE: Explanation of NeuralNetwork and Matrix implementation:

This section explains in more detail how I implemented and created the NeuralNetwork and Matrix classes. Matrices have been greatly explained in the 'Research of algorithmic methods' section in 'Analysis'. They are essentially a 2D array with some additional functions I have added that make them useful in this program.

My implementation of a neural network makes use of the Matrix class I created. A neural network consists of several 'weight matrices'. Each weight matrix takes an input and scales it by the weights in the weight matrix and then outputs this scaled result to be input into the next weight matrix. This is vaguely modelled after the brain – each input in each layer can be called a 'neuron' and the connections between neurons are called the weights. The process of inputting data into these neurons and having them pass through the weight matrices is called 'feeding-forward' – the data is fed through the neural network to produce an output. The output is used to determine the direction the snake will move in.

Each layer of neurons that is passed from one weight matrix to the next also has a 'bias neuron' which is an additional neuron that always has a value of 1.0 which can then be scaled appropriately by the weight matrix. The details as to why these are required are complicated but note that they improve the effectiveness of the learning process. I created a function in my Matrix class that adds a bias neuron to make implementation of the neural network easier.

Prototype Two:

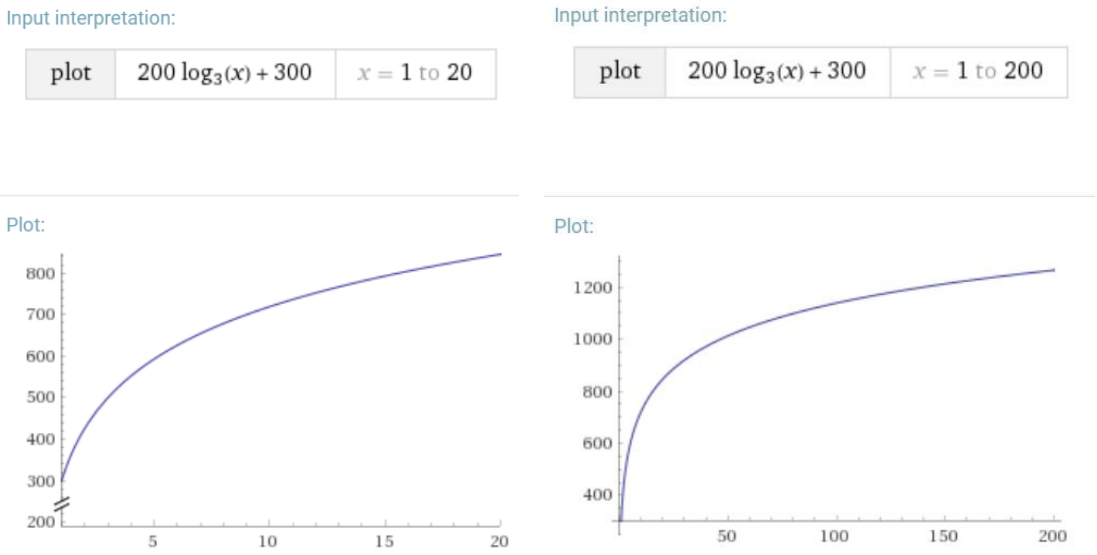
At the start of the final program, all weight matrices in the neural networks of all snakes will be randomised, meaning they will not be able to play Snake immediately as it is unlikely the random weights in the neural network will process the snake's inputs effectively. Some snakes and their neural networks will perform marginally better than the other snakes in the same generation and the next generation of snakes will learn from these better snakes. After many generations of learning from the previous generation's best snake, the snakes and their neural networks will slowly learn how to play Snake.

Log 4 – March 10th:

I need to make sure the snakes cannot be stuck in an infinite loop as they may respond to inputs in a way that causes them to not eat an apple, but also not hit any walls. Instead of detecting looped movement, I will implement a function that determines the number of allowed moves the snake can make to get the next apple. If they don't meet this amount, they will be killed.

The function I use to determine how many moves the snake can make to get to the next apple must increase over time, but the rate at which it increases should slow down. This would draw a graph that levels off. 300 movements should be a good amount for the snake to reach the first apple and a scaled base 3 logarithmic function in which the snake's length is the input should meet the requirements I described.

A visualisation of this function is below:



The graph on the left shows this graph from a snake length of 1 to 20. At first the snake has around 300 moves, but it is given 800 moves when it has a length of 20, giving it ample time to get the next apple. The graph on the right shows the same graph but over a greater domain. You can see at a length of 200, the snake will have roughly 1265 moves to get the next apple.

Log 5 – March 11th:

I have added an extra attribute to the Level class called nextAppleMoves. This holds the current number of moves allowed to get to the next apple (explained in previous log) and is only updated when the snake grows, rather than every frame.

Log 6 – March 14th:

I have removed the attribute bestIndex from the Population class as I no longer need it. Instead I use the getBest() method when I need to find the best-performing snake.

Prototype Two:

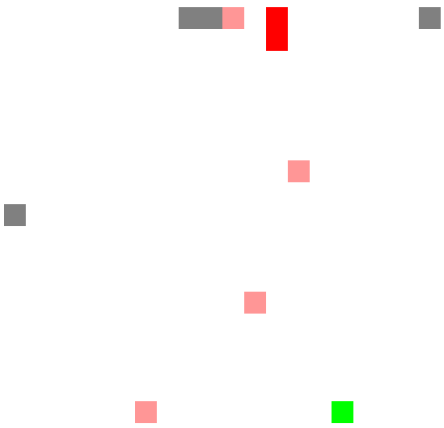
Log 7 – March 14th:

I've added a temporary attribute to the population class that stores the average score of the snakes so I can verify if they are learning or not. This has not been added to the class diagram or explained in further detail as it will be discarded after prototype 2.

Log 8 – March 14th:

I have heavily changed how the Population.show() method works. It will now sort all players in the players array based on their current score and then show the top performing snakes, rather than random snakes. The number of snakes rendered depends on the value playersRendered in the Main class. As sorting the list costs a lot of performance, it will only sort once every 200 frames, ensuring it is not too taxing on the program, whilst still regularly updating the best snakes. This requires a new attribute in the Population class called framesSinceLastSort to only allow sorting every 200 frames. Doing this means I must create a custom Comparator class in the Population class that informs how the Player array should correctly be sorted when using Arrays.sort(). This class is called PlayerComparator and includes code for comparing two players.

On top of this, I have also removed the grid lattice showing the boundaries of where the snakes can move, as I personally think it looks cleaner. After consulting with the initial focus group, both agreed it looked nicer without. Moreover, it will improve performance as there is less to draw. This is what it now looks like:



You can see that there are no longer any grid lines being displayed, which makes the program user interface cleaner.

Log 9 – March 15th:

I have changed the rendering mode on the Processing program from the default rendering mode to FX2D. This change is made in the size() function within Main.setup(). This changes the method that Processing uses to render objects in my project, massively improving rendering performance and allowing many more snakes to be drawn at once.

Log 10 – March 17th:

There is no guarantee that the next generation will improve from the last generation in the learning process, so to make this more likely, I will clone the best player from the last generation and move them straight into the next generation without mutation.

Prototype Two:

Log 11 – March 18th:

I have removed the isBest attribute from the Level class and instead changed the Player.show() and Level.show() methods to take a Boolean attribute that is true if the snake is to be rendered as the best snake or false if it is a regular snake.

Testing:

Prototype 2 involves a heavy amount of data processing – most notably in the Matrix and NeuralNetwork class. All test data used is referenced in the respective test result section. This changes the style of testing I will perform from the subjective and user-testing style in Prototype 1 to a more objective manner.

Test No.	Test Description	Expected Outcome	Actual Outcome
1	I am testing the constructor and randomize() method in the Matrix class.	After creating a new matrix and randomizing it, all values in the matrix should be randomized between -1 and 1.	As expected; see test result 1.
2	I am testing Matrix.addBias() when the dimensions are correct and incorrect.	When there is one column, the bias node should be added, but when there is more than one column, the program should throw an exception.	As expected; see result 2.
3	I am testing Matrix.mutate() with varying mutationRate values.	After mutating the matrix, some values should be different, but not dramatically.	As expected; see result 3.
4	I am testing Matrix.multiply() with multiple matrices.	When multiplying matrices, my program should produce the correct resultant matrix. It should also throw an error if the matrices cannot be multiplied.	As expected; see result 4.
5	I am testing Matrix.applySigmoid() on a matrix.	I will apply this function on a matrix of varying values and inspect the result it gives. All values should be within the range 0-1, with extreme negative values approaching 0 and extreme positive values approaching 1.	As expected; see result 5.
6	I am testing Matrix.toArray().	I will create a matrix with custom values and I expect this function to return a 1D array that retains all elements.	As expected; see result 6.
7	I am testing the NeuralNetwork constructor and feedForward() method.	I will create a new NN object and attempt to feed forward some values. I expect a random output between 0 and 1 with no errors.	As expected; see result 7.
8	I am testing Level.vision() and Level.snakeLook()	I will run the playable snake game but the array returned by Level.vision() will be printed to console. The values printed as I play should make it clear that the function is working as intended. I will take a screenshot of the game and the array returned to look at the results.	Not as expected; see result 8.
9	I am testing that Player, Population and Main classes all work together to allow the snakes to learn.	All code relating to the genetic algorithm and the AI learning how to play the game is being tested. The snakes are expected to gradually get better at the game over time.	As expected; See result 9.
10	I am testing the ability to sort the players array in the Population class.	Sorting Player objects requires a custom comparator class, which needs to be tested. I will output the result of sorting the players and check they are sorted correctly.	As expected; see result 10.

Prototype Two:

Test result 1:

To perform this test, I used the following code:

```
Matrix m = new Matrix(10, 5).randomize();
println(m.rows + "x" + m.cols);
println(Arrays.deepToString(m.data));
```

This creates a new matrix with dimensions 10 by 5 and randomizes its data. It then prints the dimensions to console, along with all the data in the matrix. Below is the matrix data written to console after running the program:

```
10x5
[[0.62471163, -0.41389596, -0.9121561, 0.009584427, -0.64086914], [-0.5666666, -0.13350725, -0.8345002,
-0.25251377, 0.43016148], [0.87847304, 0.5798116, -0.078632355, -0.16096604, -0.0036816597], [0.7918918,
0.48573744, -0.6130252, -0.67010546, -0.13812888], [-0.0936259, -0.935477, 0.7753235, 0.8147365,
-0.37914252], [-0.23413444, -0.009786963, 0.30919015, -0.053539753, 0.85749865], [0.31221402, 0.27171385,
0.63749444, -0.044647098, 0.6591054], [0.52611375, 0.90755486, 0.5954486, -0.4275365, 0.11621857],
[-0.21307337, -0.9050621, -0.34519696, 0.4491278, -0.22839582], [0.119243145, 0.3444848, 0.6194813,
0.3086418, 0.60354185]]
```

It's clear to see that every value in the matrix is randomized between -1 and 1 – hence the test is successful.

Test result 2:

I used the following code for testing erroneous data:

```
Matrix m = new Matrix(10, 5).addBias();
println(m.rows + "x" + m.cols);
println(Arrays.deepToString(m.data));
```

This produced this result – throwing an exception as expected:

```
49      else {
50          throw new IllegalArgumentException();
51      }
<
IllegalArgumentException
```

I next used this code to test valid data:

```
Matrix m = new Matrix(10, 1).addBias();
println(m.rows + "x" + m.cols);
println(Arrays.deepToString(m.data));
```

This is the result:

```
11x1
[[0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [1.0]]
```

As you can see, the matrix has dimensions 11x1, meaning there are 11 rows, rather than the initial 10 as the bias node has been added with a value of 1.0 at the end of the matrix, as expected.

Test result 3:

I used the following code for this test, with a mutationRate of 0.02:

```
Matrix m = new Matrix(10, 10).mutate();
println(m.rows + "x" + m.cols);
println(Arrays.deepToString(m.data));
```

This is the result:

Prototype Two:

```
10x10
[[0.0, 0.0, 0.0, 0.0, 0.0, -0.05877142, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
-0.06593661, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0]]
```

As you can see there are two values that are slightly different. This perfectly fits with the mutationRate of 2% as there are two values in one hundred that have been mutated.

After changing the mutation rate to 0.5, these are the results:

```
10x10
[[0.027769517, 0.0, -0.17793855, 0.0, -0.11817603, 0.0, 0.0, 0.27895898, 0.14801654, 0.03454493], [0.0,
0.24493253, 0.06464078, 0.0, 0.0, 0.0, -0.1895794, 0.019954648, 0.001282774, 0.0], [0.0, 0.0, 0.0,
-0.1131163, 0.53046834, 0.0, -0.13443622, 0.0, 0.13819622, -0.08466072], [0.073126435, -0.068622045, 0.0,
-0.18243335, -0.22636259, -0.57624495, 0.002956801, -0.124065414, 0.0, 0.4127447], [0.019298052, 0.0,
-0.036110364, -0.03127122, 0.0, 0.015299445, 0.0, 0.0, 0.0, -0.17870782], [-0.047515817, 0.0, 0.12401494,
0.055173494, 0.0, 0.0, -0.12200574, 0.25584504, -0.41724628, 0.0], [0.19185372, 0.0, 0.0, -0.056284945,
0.12776864, 0.0, -0.16507672, 0.11620073, 0.0, 0.0772551], [0.0, -0.20517817, -0.2581564, 0.0, 0.0,
0.1551393, 0.0, -0.05405755, 0.085582, 0.0], [0.01944506, 0.017760156, 0.00119922, 0.0, 0.0, -0.029045334,
0.01041189, 0.0, 0.09962518, 0.0], [0.0, 0.0, 0.0, -0.18486443, -0.14922546, 0.0, 0.0, -0.0062924623, 0.0,
0.0]]
```

From inspection, it looks as if roughly 50% are non-zero values, suggesting this test is successful. Additionally, the greatest mutation is around 0.6 away from 0 which is not too dramatic of a mutation – this confirms that the 'randomGaussian() / 5' line of code in Matrix.mutate() should be significant enough to make a small difference, but not enough to completely change a weight from its original value.

Test result 4:

I used the following code for the test to create two random 5x5 matrices and multiply them and print the result:

```
Matrix m1 = new Matrix(5, 5).randomize();
Matrix m2 = new Matrix(5, 5).randomize();
Matrix r = m1.multiply(m2);
println("m1: " + m1.rows + "x" + m1.cols);
println(Arrays.deepToString(m1.data));
println("m2: " + m2.rows + "x" + m2.cols);
println(Arrays.deepToString(m2.data));
println("result: " + r.rows + "x" + r.cols);
println(Arrays.deepToString(r.data));
```

To verify the result I get, I am copying the two matrices into a known-working system and checking it produces the same result.

```
m1: 5x5
[[0.0088477135, 0.3621049, -0.54626024, -0.16388142, -0.75359285], [-0.09649277, -0.7080507, -0.24441135,
0.70170164, -0.8161384], [0.76105464, -0.7774855, -0.6751113, 0.08300805, -0.45860255], [-0.5193727,
0.82535493, -0.40051258, -0.7468519, -0.9661881], [0.8525661, -0.60642123, -0.25342786, 0.18287015,
-0.40264523]]
m2: 5x5
[[-0.05708289, -0.9558954, 0.52222943, 0.15171039, 0.89505327], [0.8816639, 0.72776484, -0.47161973,
0.16870594, 0.83137214], [0.04615605, 0.81471527, -0.41190183, 0.37329185, -0.10985756], [0.14516163,
-0.32943892, 0.8500229, -0.6688421, -0.81818306], [0.87067354, -0.40365458, 0.6959083, 0.7819736,
0.7649673]]
result: 5x5
[[-0.3863861, 0.16820331, -0.60488415, -0.6211619, -0.07341498], [-1.2387657, -0.5239129, 0.41271782,
-1.3328543, -1.846609], [-1.1473281, -1.6855679, 0.7936157, -0.68185437, -0.30976102], [-0.21080172,
1.4068749, -1.8027331, -0.34506762, 0.13726944], [-0.9190508, -1.3604827, 0.7108624, -0.50473547,
-0.17086034]]
```

Prototype Two:

By inspection you can see the result above is equal to the verified result below:

$$C = \begin{pmatrix} -0.386386098145917615 & 0.1682032904253027 & -0.604884165299702495 & -0.621161878717222735 & -0.073414968461961255 \\ -1.238765673897055 & -0.5239128957974013 & 0.4127177744806964 & -1.3328543608998198 & -1.8466089437565883 \\ -1.1473280891613301 & -1.685567787730154 & 0.7936156347968292 & -0.6818544374726504 & -0.3097609944271172 \\ -0.21080179000245 & 1.4068748936795906 & -1.8027330497390485 & -0.3450676311587118 & 0.13726946647771 \\ -0.9190506912820487 & -1.3604825635065 & 0.7108623822202007 & -0.5047354779915112 & -0.1708603670554016 \end{pmatrix}$$

This should therefore work with all other valid multiplications, but I have tested a multiplication that shouldn't work below, with a 5x5 and a 4x4 matrix. This threw the exception below, which is intended.

```
    else {
      throw new IllegalArgumentException();
    }
```

Could not run the sketch.

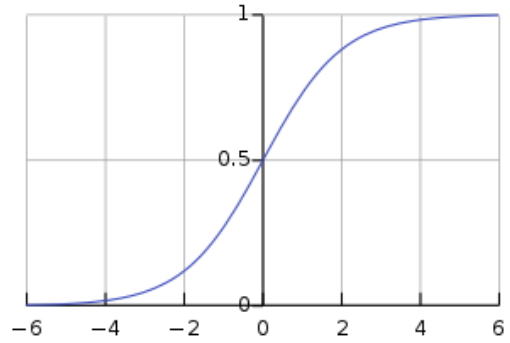
Test result 5:

This is the code I used for this test:

```
Matrix m = new Matrix(3, 4);
m.data = new float[][] {{ 0, 100000, 123, 10 }, { -102312, -10, -1000, -123 }, { -1, 1, 2, -2}};
println(m.rows + "x" + m.cols);
println(Arrays.deepToString(m.applySigmoid().data));
```

I manually created the data array for this matrix (see above), inputting a range of values – some extreme and others small. Below is the result of the program:

```
3x4
[[0.5, 1.0, 1.0, 0.9999546], [0.0, 4.5397872E-5, 0.0, 0.0], [0.26894143, 0.7310586, 0.880797, 0.11920292]]
```



This is the sigmoid function mapped on an x-y plane.

As you can see, the sigmoid function works correctly. 0 clearly maps to 0.5 after applying it to the first value and very positive values map to 1, whereas very negative values map to 0.

Test result 6:

This is the code I used for this test:

```
Matrix m = new Matrix(3, 4);
m.data = new float[][] {{ 0, 100000, 123, 10 }, { -102312, -10, -1000, -123 }, { -1, 1, 2, -2}};
println(Arrays.toString(m.toArray()));
```

This was the result:

```
[0.0, 100000.0, 123.0, 10.0, -102312.0, -10.0, -1000.0, -123.0, -1.0, 1.0, 2.0, -2.0]
```

This is successful as there are 12 elements, as expected, in a 1D array. The original order is also preserved.

James Ball

Prototype Two:

Test result 7:

This is the code for this test. There are 8 inputs as the networkStructure float array is { 8, 16, 2 }:

```
NeuralNetwork nn = new NeuralNetwork();
println(Arrays.toString(nn.feedForward(new float[] { 0, 0, 0, 0, 0, 0, 0, 0 })));
```

This is the result:

```
[0.70691603, 0.6865867]
```

This means that after the inputs were fed through the array, they produced the two outputs seen above, which is correct. The values also lie in the range 0.0-1.0 which was expected.

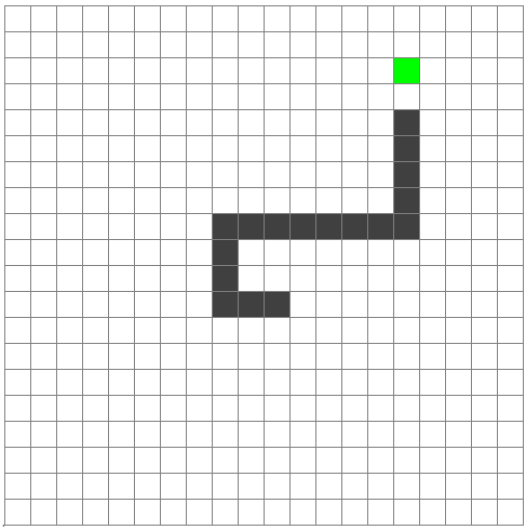
I re-ran the program with a networkStructure of { 4, 16, 24, 98, 5 } and randomized inputs to really test if it works as it should and it produced the following results:

```
[0.01851141, 0.81155723, 0.5152342, 0.25989902, 0.40632346]
```

There were no errors in any test and the 5 outputs were random, so it is working as expected.

Test result 8:

After controlling the snake manually and screenshotting a frame and its respective vision output from the program, I noticed that the results for the distance to the snake's body and the apple were always 0:



```
[0.0, 0.0, 0.06666667, 0.0, 0.0, 0.16666667, 0.0, 0.0, 0.16666667, 0.0, 0.0, 0.16666667, 0.0, 0.0, 0.125,
0.0, 0.0, 0.125, 0.0, 0.0, 0.05882353, 0.0, 0.0, 0.06666667]
```

Each of the eight directions make up three values in this array, so the first value in each group of three is the distance to the snake's body (if it is in that direction) and the second value is the distance to the apple. It is clear to see from the screenshot that ahead of the snake is the apple, but all second values are equal to 0, so this test is unsuccessful.

After inspecting the code, I realised the mistake and you can see the changes I made below:

Before:

```
if (grid[(int)pos.x][(int)pos.y] == SNAKE && vision[0] != 0) {
  vision[0] = 1.0 / (float)dist;
}

if (grid[(int)pos.x][(int)pos.y] == APPLE && vision[1] != 0) {
  vision[1] = 1.0;
}
```

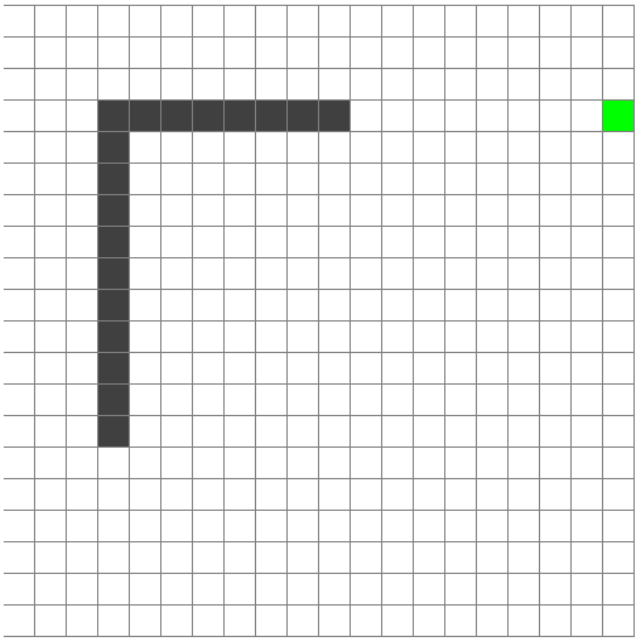
Prototype Two:

After:

```
if (grid[(int)pos.x][(int)pos.y] == SNAKE && vision[0] == 0) {
  vision[0] = 1.0 / (float)dist;
}

if (grid[(int)pos.x][(int)pos.y] == APPLE && vision[1] == 0) {
  vision[1] = 1.0;
}
```

I changed the != to a == in the if statement as before the apple and snake vision values were never updated. These are the results after re-running:



```
[0.0, 0.0, 0.055555556, 0.0, 0.0, 0.09090909, 0.0, 1.0, 0.09090909, 0.0, 0.0, 0.2, 0.0, 0.0, 0.2, 0.0,
0.0, 0.2, 0.5, 0.0, 0.083333336, 0.125, 0.0, 0.083333336]
```

This now works as expected. The 8th value in the array corresponds to the apple vision value for the direction is currently facing, meaning the snake can see that there is an apple in front of it. Distance to the snake's own body is also shown now.

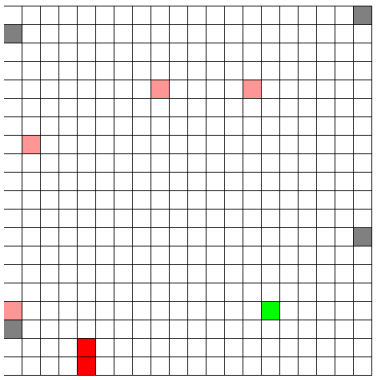
Test result 9:

There is no specific 'test code' for this test as it is all the code involved in the learning process of the snakes. This involves the Player Level, Population and Main class all working together.

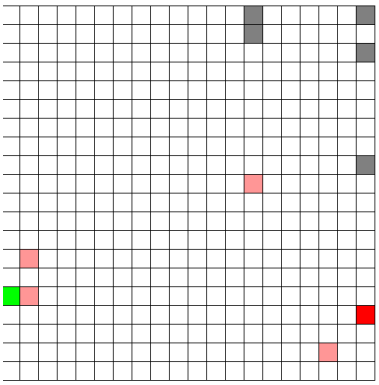
The screenshots below show the training process of the snake. The current top five snakes out of the total population are displayed. The current generation, the number of dead snakes in this generation and the average score of the last generation are all displayed on the right of the grid.

Prototype Two:

Initially, the population starts out with an average score of 0.052, but after 20 generations, this more than triples to 0.178.

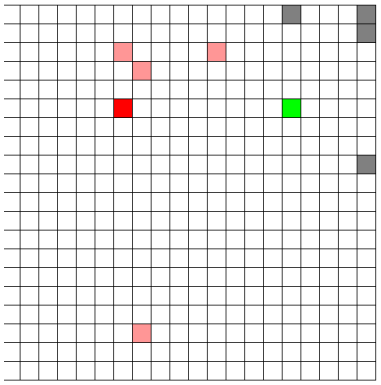


Generation: 1
Number dead: 461
Average score: 0.052

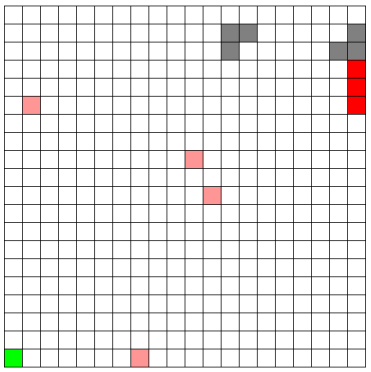


Generation: 20
Number dead: 452
Average score: 0.178

At generation 90, the average score is now 0.246, however, after over 450 generations of snakes, the average score stagnates at around this average. There is clear progress of the snakes learning initially, but they are not taking the next step to increase their average score.



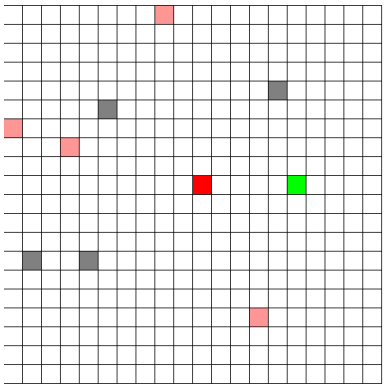
Generation: 90
Number dead: 293
Average score: 0.246



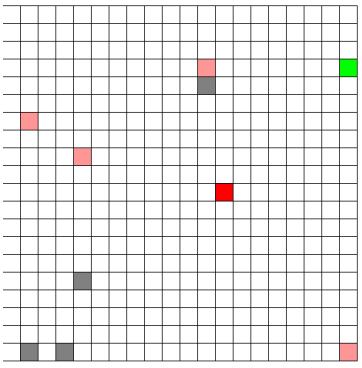
Generation: 459
Number dead: 414
Average score: 0.234

I decided to run the program again, and got dramatically different results:

After 71 generations, the average of 0.366 is already higher than the stagnated average from the first test and roughly 20 generations later, the average increases to 0.494.



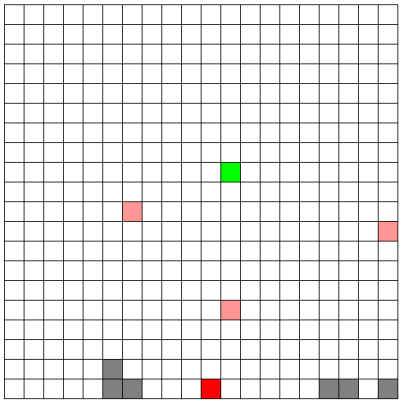
Generation: 71
Number dead: 31
Average score: 0.366



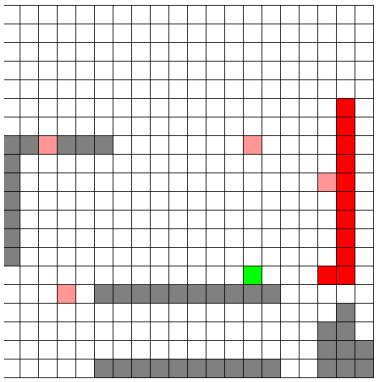
Generation: 87
Number dead: 457
Average score: 0.494

The stagnation issue appears to happen again from generation 127 to around 470, with an average of 0.65 roughly, but at 586 the average increases dramatically to over 1. You can also see on the right that the best snakes are of a considerable length and are starting to turn and move like real players, rather than the dots on the left.

Prototype Two:

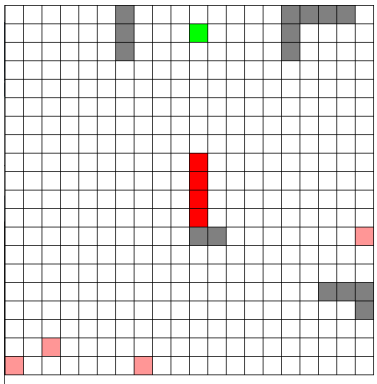


Generation: 127
Number dead: 477
Average score: 0.656

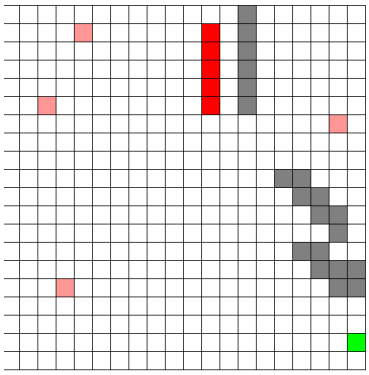


Generation: 486
Number dead: 294
Average score: 1.002

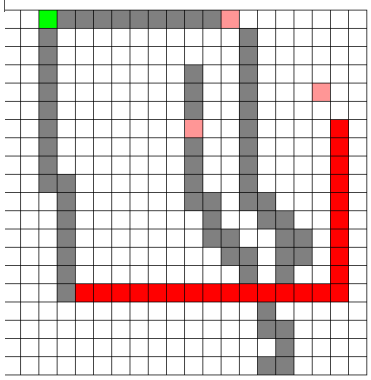
Between roughly generation 470 to 540, the average score of the snake rapidly increased, as can be seen from these screenshots.



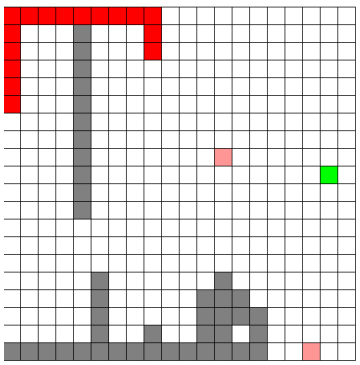
Generation: 494
Number dead: 163
Average score: 1.822



Generation: 496
Number dead: 208
Average score: 2.46

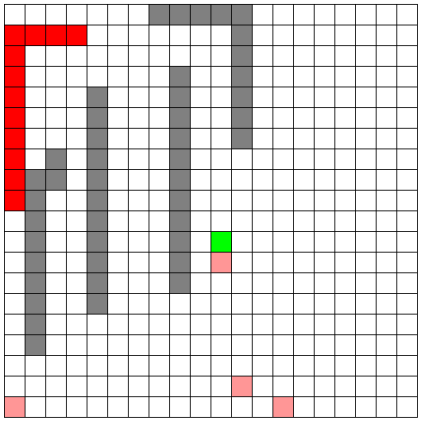


Generation: 505
Number dead: 473
Average score: 3.844



Generation: 521
Number dead: 179
Average score: 5.892

I stopped the program shortly after this screenshot as it was evident that the program was working correctly, although there are some issues with stagnation.



Generation: 554
Number dead: 151
Average score: 7.976

Prototype Two:

It is clear to me that every time I run the program, I am going to see dramatically different results – some populations will learn advanced movements quickly, whereas others will stagnate for a long time before learning anything significant. The code implemented clearly allows the snakes to learn, but I will look more closely at reducing the chance of the population stagnating in the 'Program optimisation' section of Prototype 3.

Test result 10:

This is the code I used to test this:

```
Arrays.sort(players, new PlayerComparator());
Collections.reverse(Arrays.asList(players));

println("START");
for (int i = 0; i < players.length; i++) {
  print(players[i].level.score + ", ");
}
println();
```

The players array is first sorted and reversed (i.e. ordered by score, descending) and then all player scores are written to console. Below is the result after running the program for a while:

```
START
19, 17, 16, 16, 16, 14, 14, 13, 13, 12, 12, 11, 11, 10, 10, 10, 10, 9, 8, 8, 7, 6, 5, 5, 4, 4, 4, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 4, 0, 3, 5,
2, 1, 1, 5, 2, 2, 3, 2, 1, 2, 6, 1, 2, 0, 3, 1, 1, 2, 2, 5, 1, 1, 0, 3, 2, 3, 1, 3, 3, 1, 1, 5, 1, 3, 2, 5, 5, 5, 1, 3, 4, 3, 4, 2, 0, 1, 0, 5, 1, 2, 3, 0, 1,
1, 1, 1, 3, 1, 1, 6, 2, 2, 0, 1, 3, 1, 2, 1, 1, 2, 3, 5, 1, 5, 2, 3, 1, 2, 2, 4, 6, 1, 3, 3, 1, 1, 4, 3, 1, 1, 4, 1, 1, 2, 3, 3, 0, 3, 5, 2, 1, 4, 5, 0, 4, 1,
0, 8, 2, 6, 1, 3, 4, 1, 1, 6, 3, 2, 3, 0, 0, 3, 3, 1, 6, 5, 5, 1, 2, 3, 3, 2, 3, 0, 3, 0, 6, 2, 4, 5, 5, 1, 6, 2, 4, 6, 5, 3, 0, 2, 0, 3, 0, 2, 3, 1, 1, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 12, 14, 4, 4, 9, 7, 11, 9, 10, 8, 9, 8, 10, 8, 7, 5, 10, 6, 9, 7, 7, 10, 8, 9, 9, 7, 10,
8, 5, 5, 7, 13, 7, 13, 7, 8, 9, 7, 11, 9, 6, 14, 9, 6, 9, 7, 15, 10, 9, 9, 9, 8, 13, 15, 9, 13, 10, 10, 12, 15, 13,
```

You can see that at first glance it only seems to have partially sorted the list, but this is the intended result, as all snakes that have already died will effectively have a lower score when being compared to a snake that is still alive. This means that all the snakes that are still alive are sorted correctly at the beginning of the list. I will only be showing snakes that are alive and the players array is sorted to show the best snakes, so the order of dead snakes in the array is arbitrary.

This is the intended result and is therefore a success.

General outcome of testing and user feedback:

All but one tests were successful, with the single failed test now being resolved. Test 9 details the most important test in this prototype as it tests if the snakes can learn how to play Snake as expected. As this was proven to be successful, testing in general was also successful, and there is nothing to fix before the next prototype as a result.

After meeting with the original focus group, they agreed that the program performs as expected. The undergraduate student mentioned that the statistics I included, such as the number of dead snakes and the average score of the last generation, were very interesting as they showed the different learning processes the snakes take as they improve. He noted that once the snakes learn something new, the average score rapidly increases and then starts to level off again once they finish learning. The younger student said he could clearly see the snakes beginning to learn and get longer as generations passed.

It is evident from the inconsistent training results displayed in test 9, I will also need to focus on finding the right hyper-parameters to use for optimal learning performance in the next prototype. This includes the mutation rate, neural network structure, activation function and population size.

Evaluation:

In this section I will compare the finished prototype to its success criteria detailed in the Analysis section.

1. A system for choosing the best performing snakes randomly must be implemented.

   This criterion is perfectly met in Population.selectParent(). This function selects a player randomly but the chance of a player being chosen is directly proportional to the player's score squared. This makes it increasingly more likely for a player to be chosen if they have a higher score than other players.

   As this is a semi-random process, there is also a chance that worse snakes move to the next generation, which meets the description of success criteria 1.

2. Mutation must be introduced in each genome after each generation.

   When the next generation of players is being created in the Population.naturalSelection() procedure, all players are mutated before being added to the next generation. This is achieved using the NeuralNetwork.mutateWeights() function which slightly mutates a random amount of weights, determined by the mutationRate attribute in Main.

   This meets the needs of success criteria 2 by adding another element of randomness as seen in biological evolution.

3. A system for crossover between two genomes should be implemented.

   Another function, Population.uniformCrossover(), fits this criterion. This function crosses over two chosen parents and produces a child neural network that shares roughly half of its weights with one parent and the rest with the other.

   This function is used in Population.naturalSelection() when new players are being created for the next generation. It adds a further layer of randomness.

4. A method of easily creating neural networks of different sizes should be present.

   The NeuralNetwork class has been specifically coded in a way to ensure creating NNs of different sizes is very easy. I can easily change how many layers there are in the NN and how many nodes are in any one of those layers.

   This is evidenced by the networkStructure array in the Main class. This is an integer array that determines the number of layers and the dimensions of each layer in the neural network. Changing the values of this array changes the size of the NN. This is exactly what I specified in the description of success criteria 4.

   This will allow me to test various neural network structures in Prototype 3.

5. I need to create conditions that determine the death of each snake.

   This criterion addresses the issue of snakes being stuck in loops, which I talk about in detail in development logs 4 and 5. These logs detail the process by which snakes are killed if they are not making any progress.

   Instead of detecting when a snake is in a loop, there is an allotted amount of time given to each snake to get to the next apple. If they don't meet this time, they are killed. The amount of time

given increases based on the length of the snake, as it will take longer to get to each apple as the snakes get longer.

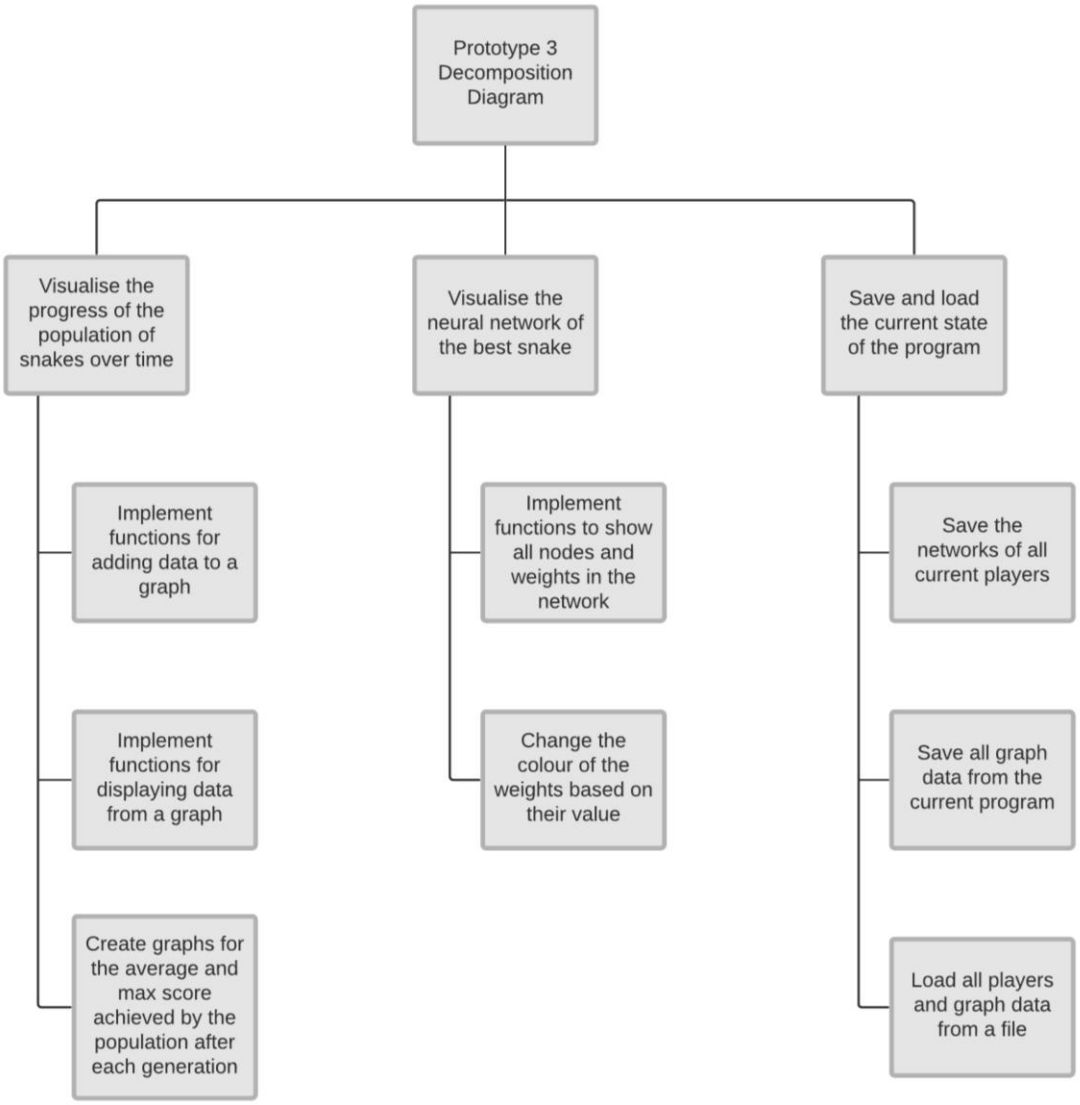
This stops a snake that is stuck in a loop and means the program will never be stopped by a snake that is stuck. This means this criterion is met.

This evidences that I have met all the success criteria I detailed in the Analysis section.

Prototype Three:

This prototype adds graphs to show the maximum and average score of the snake population, as well as a graph showing the neural network of the best snake. I also need to implement a system for saving the current state of the program, so progress can continue after closing the program. I will also be spending much of this prototype experimenting with the different hyper-parameters of the program (e.g. mutation rate, neural network structure etc.).

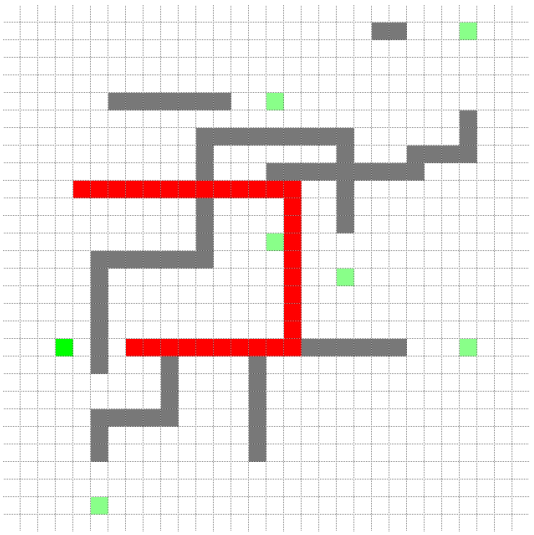
Decomposition Diagram:



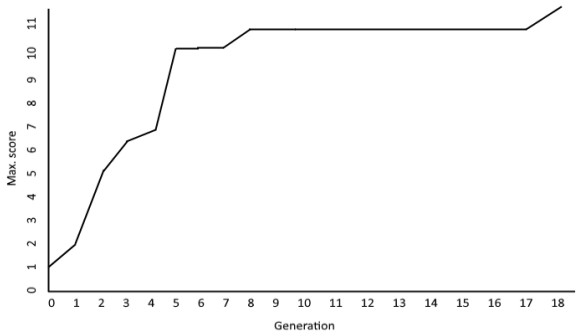
This demonstrates the sub-problems that prototype 3 is to be broken-down into. The main parts of this prototype are to visualise the average and maximum score graphs of each generation, to visualise the best neural network and to save and load the current state of the program. Each sub-problem is further broken down into further sub-problems.

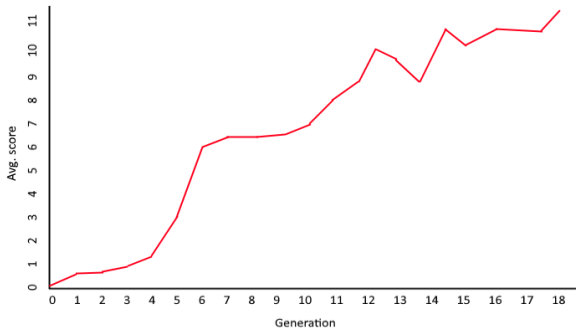
Prototype Three:

Initial sketch:

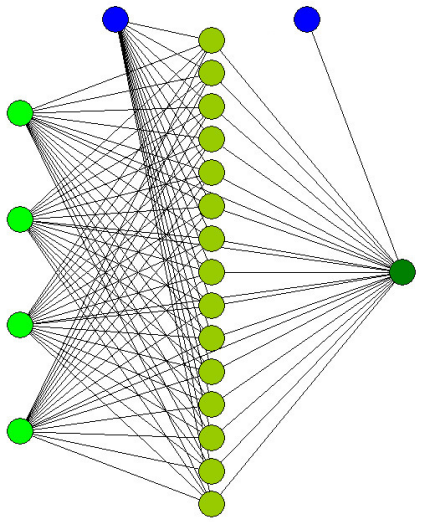


Players rendered: 7
Number dead: 350
Framerate: 235
Score of best: 23





(Neural network image found online to use/modify freely)



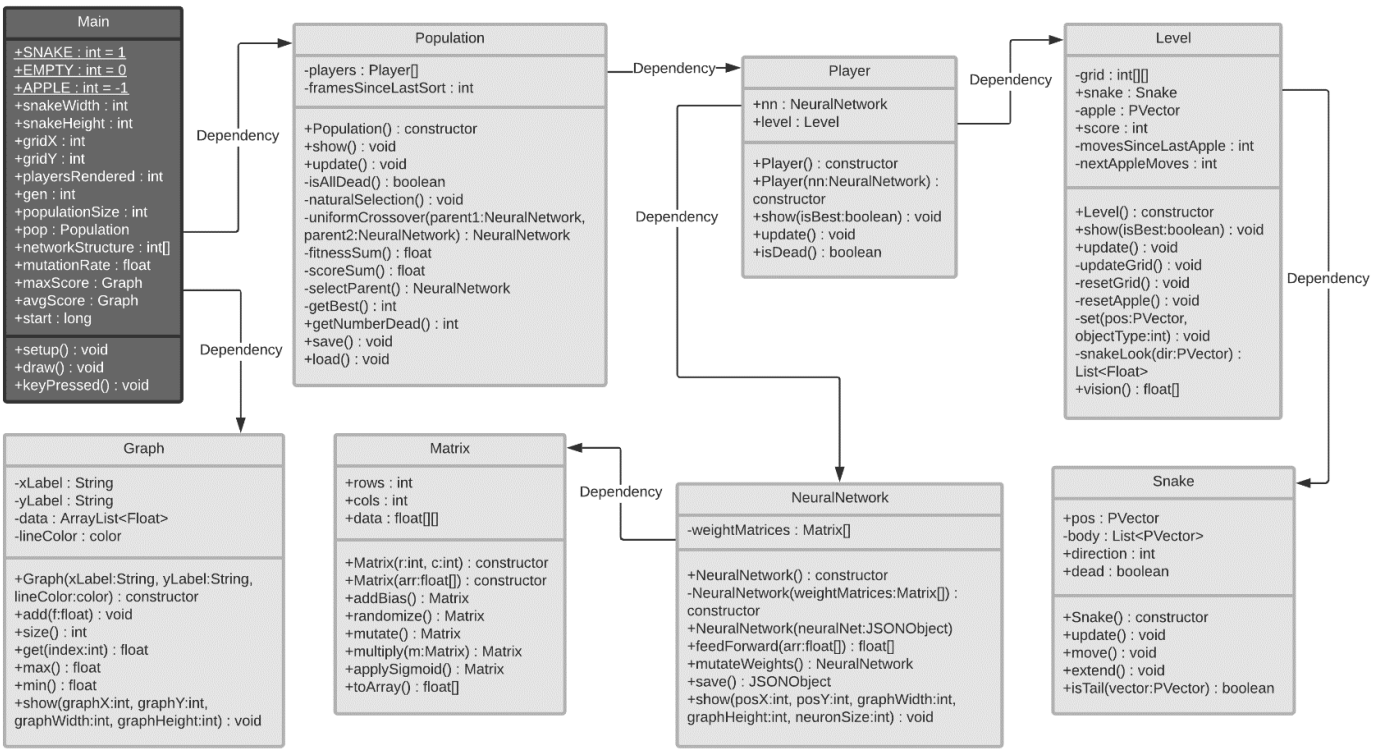
This sketch builds upon the sketch in Prototype 2 by adding graphs to show the scores of the population as generations pass and the neural network of the best snake. Something not displayed in the neural network graph is the colour-coded weights. I plan to make the weight lines green if the weight is positive or red if the weight is negative.

The neural network graph will support the modularity of the NeuralNetwork class, meaning it will support a changing structure of the neural network, with any number of layers and neurons.

Prototype Three:

Classes:

Class Diagram:



All classes, including those introduced in earlier prototypes, are included in this diagram. The Graph class has been added after Prototype 2 and the Main, Population and NeuralNetwork classes have been modified from Prototype 2.

Explanation of key algorithms:

NeuralNetwork.show():

Taking the x position, y position, width, height and neuron size as parameters, this procedure is responsible with visually representing the matrices and neurons in a NeuralNetwork object. The pseudocode is below:

```
public procedure show(int posX, int posY, int graphWidth, int graphHeight, int neuronSize)
    int layerAmount = networkStructure.length

    Vector[][] neuronPos = new Vector[layerAmount][]

    for (int i = 0, i < layerAmount - 1, i++)
        // creates vector arrays to hold all the locations that neurons will be drawn to
        // the screen. The +1 accounts for the extra bias neuron.
        neuronPos[i] = new Vector[networkStructure[i] + 1]
    end for

    // the output layer doesn't have any bias node.
    neuronPos[layerAmount - 1] = new Vector[networkStructure[layerAmount - 1]]

    // defines the pixel gap between layers in the network.
    int layerXSteps = graphWidth / (layerAmount - 1)

    for (int i = 0, i < neuronPos.length, i++)
        // the neurons in the layer is different if there is a bias node.
        int neuronsInLayer = networkStructure[i]

        // if the layer is not the output layer, there is an extra bias neuron.
```

```
        if (i != neuronPos.length - 1)
           neuronsInLayer++
        end if

        // defines the vertical pixel gap between neurons in a layer.
        int layerYSteps = graphHeight / neuronsInLayer
        // this padding centres neurons in their layer.
        int layerYPadding = (graphHeight – ((neuronsInLayer - 1) * layerYSteps)) / 2

        for (int j = 0, j < neuronPos[i].length, j++)
           // calculates the position of this neuron.
           neuronPos[i][j] = new Vector(posX + i*layerXSteps,
                             posY + layerYPadding + j*layerYSteps)
        end for
    end for

    for (int i = 0, i < weightMatrices.length, i++)
        for (int j = 0, j < weightMatrices[i].rows, j++)
           for (int k = 0, k < weightMatrices[i].cols, k++)
               float weight = weightMatrices[i].data[j][k]
               // makes the line green if the weight is positive, red if negative.
               if (weight > 0)
                  // RGB colour
                  change line colour to (0, 255*weight, 0)
               else
                  change line colour to (-255*weight, 0, 0)
               end if

               draw line from neuronPos[i][k] to neuronPos[i + 1][j]
           end for
        end for
    end for

    for (int i = 0, i < neuronPos.length, i++)
        for (int j = 0, j < neuronPos[i].length, j++)
           draw circle with centre neuronPos[i][j] and diameter neuronSize
        end for
    end for
end procedure
```

NeuralNetwork.save():

This converts all the attributes of a NeuralNetwork object into a JSONObject so that it can be easily stored to a file to be loaded later. This pseudocode is very simple, as there are a few more steps in the actual code, but this explains the logic behind the function. Pseudocode below:

```
public function save()
    // JSONFile object allows standard datatypes to be added to it so it can be saved to a
    // file.
    JSONFile neuralNet = new JSONFile()

    for (int i = 0, i < weightMatrices.length, i++)
        // appends the weight matrix 2D matrix to the JSONFile.
        neuralNet.add("weightMatrix " + i, weightMatrices[i].data)
    end for

    return neuralNet
end function
```

NeuralNetwork(JSONObject):

This is the constructor for the class that takes a JSONObject as a parameter and loads it into the NN object. Pseudocode below:

```
public constructor NeuralNetwork(JSONFile neuralNet)
    // calls the NeuralNetwork() constructor for this object.
    this = new NeuralNetwork()
```

```
    for (int i = 0, i < weightMatrices.length, i++)
        // sets this object's weightMatrices from the weightMatrices in the JSONFile.
        weightMatrices[i].data = neuralNet.get("weightMatrix " + i)
    end for
end function
```

Population.save():

The current program state is converted to a JSONObject and saved to a file. This includes the neural networks of all players in the Population class and the data displayed in the maxScore and avgScore graphs. program.add() The pseudocode for this is below:

```
public procedure save()
    sort(players, ascending)

    JSONFile program = new JSONFile()

    program.add("layerCount", networkStructure.length)

    // adds the neural network dimensions to the JSONFile object.
    for (int i = 0, i < networkStructure.length, i++)
        program.add("length " + i, networkStructure[i])
    end for

    program.add("gen", gen)

    // adds each player's neural network to the JSONFile.
    for (int i = 0, i < players.length, i++)
        program.add("neuralNetwork " + i, players[i].nn.save())
    end for

    float[] maxScoreData = new float[maxScore.size()]

    // converts the graph data into an array of floats.
    for (int i = 0, i < maxScore.size(), i++)
        maxScoreData[i] = maxScore.get(i)
    end for

    float[] avgScoreData = new float[avgScore.size()]

    for (int i = 0, i < avgScore.size(), i++)
        avgScoreData[i] = avgScore.get(i)
    end for

    program.add("maxScore", maxScoreData)
    program.add("avgScore", avgScoreData)

    saveJSONFile(program, "/data/program.json")
end procedure
```

Population.load():

This takes the JSONObject saved to a file using Population.save() and loads all the information from this file into the Population object, including the neural networks of players and all graph data. Although this is a key algorithm in this prototype, the pseudocode is self-explanatory as it is simply a reversed process to Population.save(). It involves retrieving the JSON file and then fetching all weightMatrices and graph data from it and loading it into the program.

Graph.show():

This draws a line graph of data to the screen at a specified x and y position. It is responsible for making sure the lines scale to the size of the graph and drawing axis labels. textWidth(String) returns the width in pixels of a string of text. The pseudocode is below:

Prototype Three:

```
public procedure show(int graphX, int graphY, int graphWidth, int graphHeight)
    // only draw the graph if there is at least two datapoints to draw between.
    if (size() > 1)
        float min = min()
        float max = max()
        float valueRange = max – min

        if (valueRange == 0)
           // valueRange cannot equal 0, so set it to an arbitrarily small float.
           valueRange = arbitrarily small float
        end if

        write xLabel at (graphX + 10 - textWidth(xLabel) / 2 + graphWidth / 2, graphY +
        graphHeight + 50)
        // in the Java code, this requires rotating the x-y plane. Explained in log 2.
        write yLabel vertically at (graphX - 60, graphY + textWidth(yLabel) / 2 +
        graphHeight / 2)

        // lineHeight() is a function that returns the height of a line of text (pixels).
        float textHeight = lineHeight() * 2
        float numberOfYLabels = graphHeight / textHeight
        // calculates the value by which the y-value labels increase by.
        float verticalIncrement = valueRange / numberOfYLabels

        for (int i = 0, i < numberOfYLabels, i++)
           // Draws text showing the y-axis label at the height specified.
           write (min + i*verticalIncrement) at (graphX - 48, graphY + graphHeight –
           i*textHeight + 5)
        end for

        // calculates the increment in the x-axis labels for there to be no label overlap.
        int horizontalIncrement = 1 + ((textWidth(gen.toString()) + 20)*gen) / graphWidth
        // textWidth determines the width between each value on the x-axis.
        float textWidth = graphWidth/(size()-1)

        for (int i = 0, i < size(), i += horizontalIncrement)
           write i at (i*textWidth + graphX+10 - textWidth(i.toString())/2,
           graphY+graphHeight+25
        end for

        change line colour to lineColor

        // draws line graph.
        for (int i = 0, i < size() – 1, i++)
           float value1Y = ((data.get(i) - min)*textHeight) / verticalIncrement
           float value2Y = ((data.get(i + 1) - min)*textHeight) / verticalIncrement

           // defines the x position of where the line is drawn from.
           float xPos = i*textWidth + graphX + 10

           // draws the line.
           draw line from (xPos, graphY + graphHeight - value1Y) to (xPos + textWidth,
           graphY + graphHeight - value2Y)
        end for
    end if
enf procedure
```

Prototype Three:

Main class:

The main class is fundamentally the same as previous prototypes, with the addition of maxScore and avgScore Graph objects as attributes. These are explained below.

Attributes:

- maxScore – This is a Graph object that stores the maximum score achieved for all generations. It will have a y-axis label of 'Max. score' and an x-axis label of 'Generation'.
- avgScore - This is a Graph object that stores the average score achieved for all generations. It will have a y-axis label of 'Avg. score' and an x-axis label of 'Generation'.

Graph class:

Graph is a new class that stores all attributes and methods for graph creation, editing and visualising. Its features are detailed below:

Attributes:

All attributes in this class are private.

- xLabel – This is a string that stores the text displayed underneath the horizontal axis.
- yLabel – This is a string that stores the text displayed beside the vertical axis.
- data – A list of floating-point values representing the data to be visualised by the graph.
- lineColor – A color variable (Processing data type) that stores the colour of the line to be drawn.

Methods:

- Graph(String xLabel, String yLabel, color lineColor) – This is a public constructor that takes these parameters and assigns them to the respective attribute. The data list is also initialised in this method.
- add(float f) – This public procedure adds the float passed as a parameter to the data list.
- size() – This public function returns the size of the data list.
- get(int index) – This public function returns the value stored in the data list at the index specified.
- max() – Returns the highest value in the data list.
- min() – Returns the lowest value in the data list.
- show(int graphX, int graphY, int graphWidth, int graphHeight) – This public procedure displays the graph represented by the values stored in the data list. The graph is drawn at the coordinates graphX and graphY with a width and height of graphWidth and graphHeight.

NeuralNetwork class:

NeuralNetwork has new methods for visualising the neural network as well as saving and loading the neural network to and from a file.

Methods:

- show(int graphX, int graphY, int graphWidth, int graphHeight, int neuronSize) – This displays a visual representation of the NeuralNetwork object it is called from at the specified coordinates and with the dimensions specified. Positive weights between neurons are displayed as green and negative weights are red.
- save() – Public function that converts all neural network attributes into a serializable format (JSON) and returns this as a JSONObject.
- NeuralNetwork(JSONObject neuralNet) – Public constructor that reverses the process carried out in NeuralNetwork.save() by taking a JSONObject as a parameter and fetching data from it to load into the NN object.

Prototype Three:

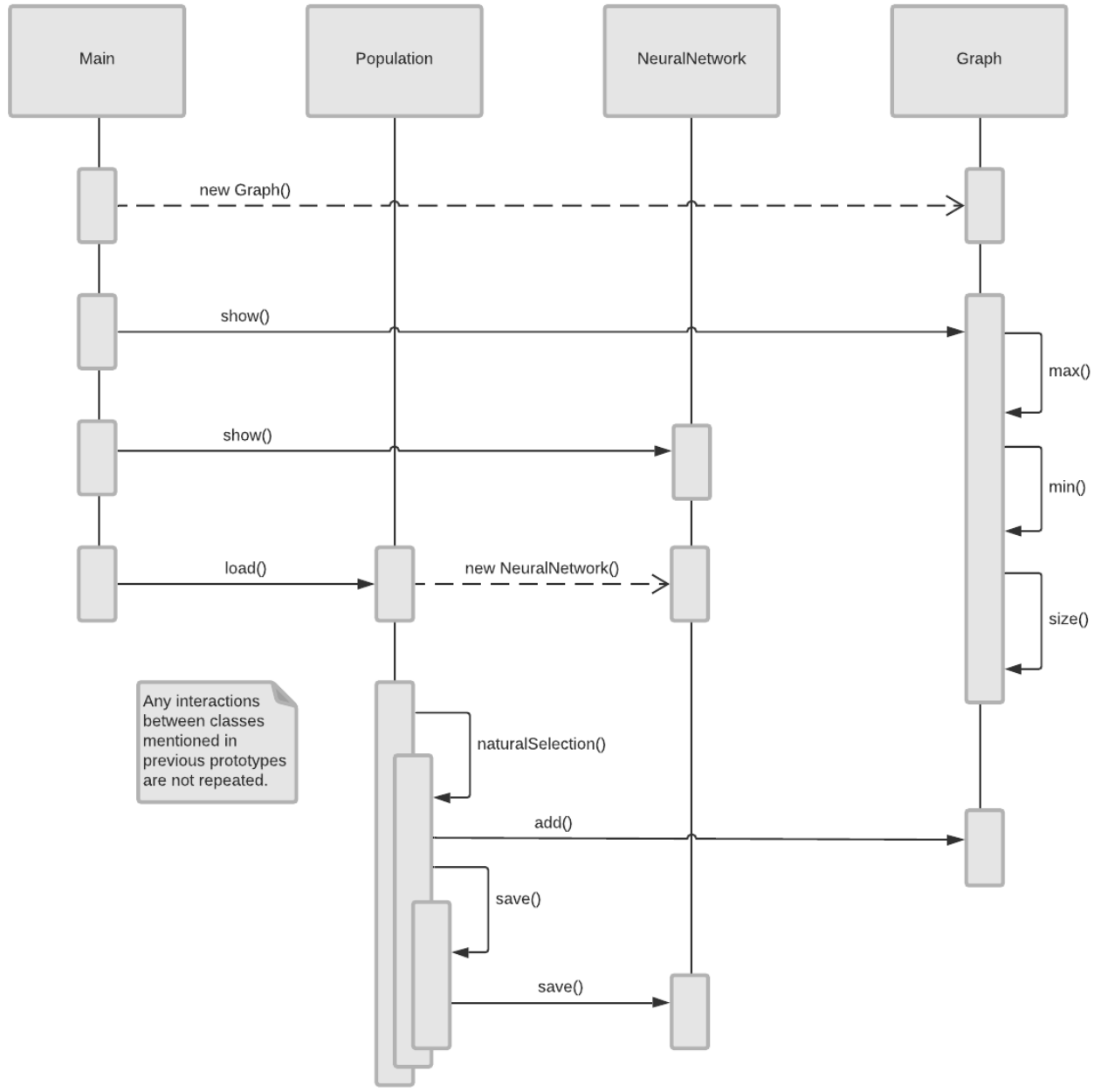
Population class:

I have made additions to this class to support saving and loading the program-state to and from a file.

Methods:

- save() – This is a public void that converts all player's neural networks and all graph data into a serializable format (JSON) so that it can be saved to a file and loaded in future executions of the program.
- load(String path) – Public void that loads a JSON file from the specified path and deserializes it so that the program saved to it is loaded into the program. This loads all graph data and player neural nets.

Sequence Diagram:



All dependencies between classes introduced in this prototype are featured in this diagram. Any dependencies in older prototypes are not repeated.

James Ball

Prototype Three:

Development Log:

Any errors I come across during development are tackled with in the Testing section, rather than in the development log as tests are performed during development. This log contains all changed functionality.

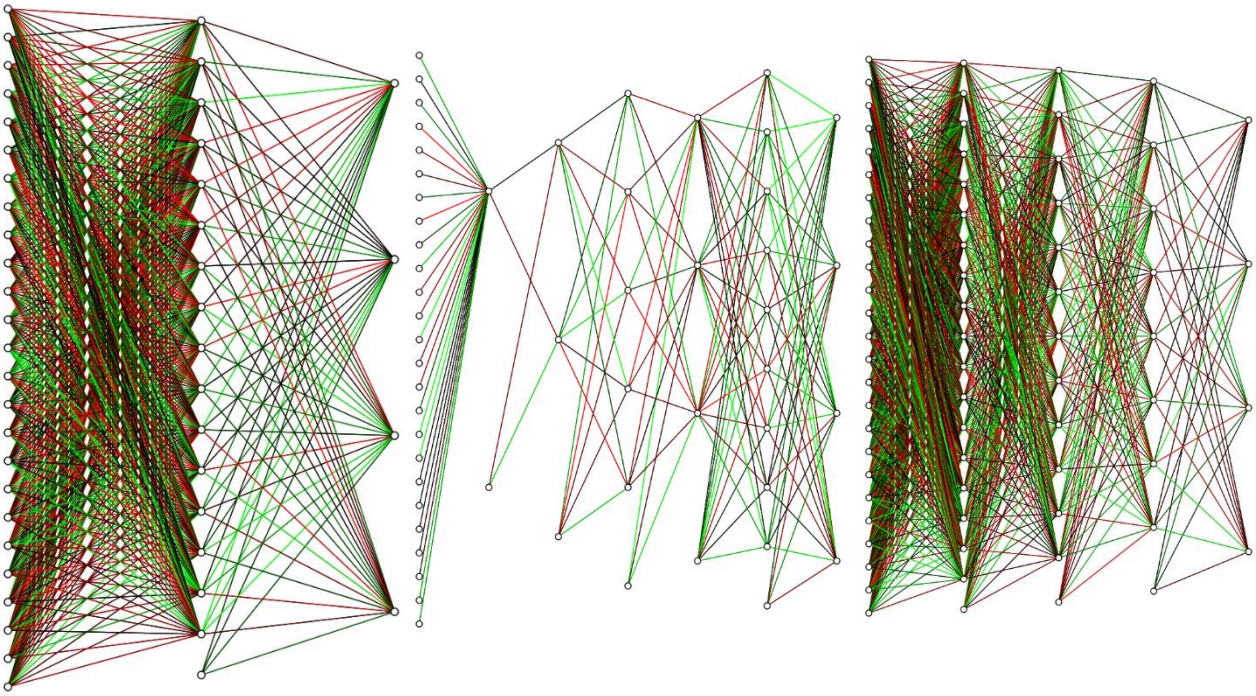
Log 1 – March 20th:

After the first test, the neural network graphs work correctly, but I need to add bias nodes to the layers in the network as this detail is currently missed. No weights go into the bias nodes, weights only go out of it, so they display differently to other neurons in the network.

I have modified NeuralNetwork.show() to account for bias nodes. This involves changing the dimensions of the neuronPos 2D array in this method to also calculate the position of bias nodes.

I then changed the triple nested loop to iterate over all columns in each weight matrix so that the bias node weights were also included.

I ran the program, and this was the result on some different structures:



The traditional structure is on the left, with an extremely unconventional structure in the middle and a deeper structure on the right. Although the middle structure looks fragmented, it renders exactly as expected and only looks fragmented due to the structure of the network. You can see bias nodes on every one of these networks (the extra node at the bottom of each layer).

Log 2 – March 20th:

When rendering text on the y-axis I need to rotate the label so that it is vertical, as demonstrated in the initial sketch for this prototype. This involves making use of a multitude of Processing's in-built functions. It requires me to rotate the x-y plane and then draw the text at this new rotation and then revert to the normal x-y plane. I have tested this functionality in Main.draw() and the code is below:

Prototype Three:

```
String label = "Vertical text.";

int x = 1400;
int y = 100 - (int)textWidth(label) / 2;

fill(0);
textSize(24);

pushMatrix();
translate(x,y);
rotate(HALF_PI);
text(label, 0, 0);
popMatrix();
```

pushMatrix() pushes the current x-y plane to a stack and popMatrix() pops it from the stack after I have drawn the text. translate() moves the origin of the plane to the x and y position specified. rotate() rotates the x-y plane so that positive x and positive y point towards a different angle. I use these in conjunction to rotate the grid about the point I want to draw the text and draw the text at this new location.

This is the result:

This works as I would like it to, but the text should be flipped so that it reads bottom to top, rather than top to bottom. After modifying the code and re-running I got the result I was looking for:

I can now use this in Graph.show() to draw a vertical label.

Prototype Three:

Log 3 – March 20th:

Code is needed to calculate the increments that the y-axis should increase by for each value label on the axis. This should react to the size of the graph, so that there are more labels as the graph's height increases. The height of the text should be calculated and used to make sure there is only a label every 3 lines of text, for example. This prevents the labels overlapping each other.

I have implemented this idea using the code below:

```
/* textHeight determines the height between each value on the y-axis. */
float textHeight = textAscent() * 3;
/* Determines the number of labels on the y-axis. */
float numberOfYLabels = graphHeight / textHeight;
/* Determines the gap between y-axis labels. */
float verticalIncrement = valueRange / numberOfYLabels;

for (int i = 0; i < numberOfYLabels + 1; i++) {
  /* Draws text showing the y-axis label at the height specified. */
  text(min + i*verticalIncrement, graphX - 48, graphY + graphHeight - i*textHeight + 5);
}
```

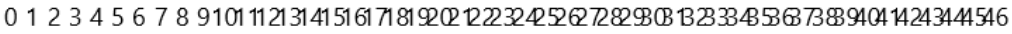
textAscent() is a Processing function that returns the height of a line of text. textHeight determines the gap between text (in this case 3*textAscent()).

The image on the left is the output it produces:

There is an equal gap between each label, going from the minimum value on the bottom of the axis to the maximum value on the top. This will scale up as the average score increases.

As I can see this works for the y-axis, I will employ a similar technique for the x-axis.

After an initial test with the x-axis, the numbers draw correctly, but gaps between labels need to be introduced to ensure there is no overlap:



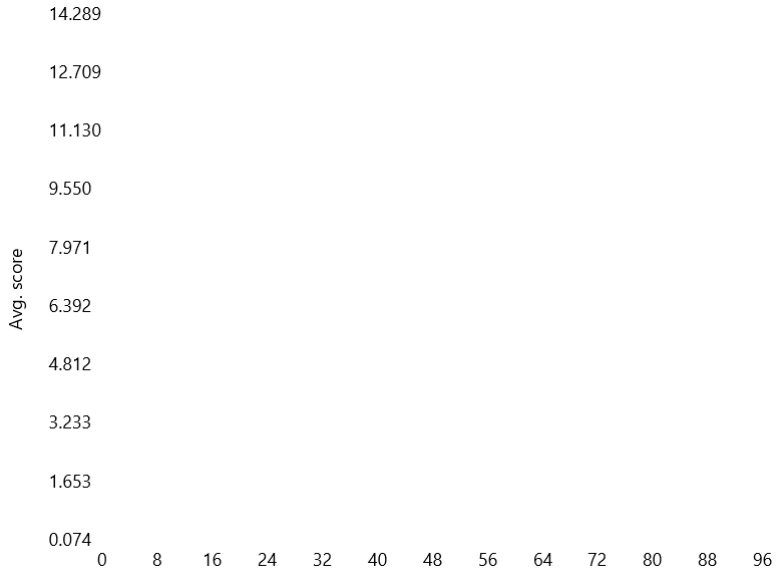
I need to use the textWidth() function that Processing provides to find the integer gap that I should take between each number on the x-axis. I stored this in a variable and used this as the increment in a for loop as seen below:

```
int horizontalIncrement = 1 + (int)((textWidth(Integer.toString(gen)) + 20)*gen) / graphWidth;

for (int i = 0; i < size(); i += horizontalIncrement) {
  text(i, i*textWidth + graphX+10 - textWidth(Integer.toString(i))/2, graphY+graphHeight+25);
}
```

Prototype Three:

This is the result after running my program:



You can see that the x-axis is incrementing in multiples of eight. The multiple it increases by changes based on how crowded the x-axis becomes. I now have an x-axis and y-axis that don't have overlapping labels and go up in sensible increments.

Log 4 – March 20th:

I am going to include a range of extra information about the currently-running program, including the average score of the last generation, the current framerate, the elapsed time since the program started, the score of the best-performing snake and the number of snakes that are being displayed to the screen.

Additionally, it should be possible to control the number of players currently displayed while the program is running. I will detect user input and if they press the + and - keys, they can change the number of players rendered.

Detecting the elapsed time since the program started requires an extra attribute in the Main class that stores the time when the program started. This is called 'start' and is a long data type.

I will also colour the text displaying last generation's average score depending on whether it is better than the generation before it or not.

The code for all this extra information is on the next page:

Prototype Three:

```
/* millis() - start is the number of milliseconds since the start of the program. */
long seconds = (millis() - start) / 1000;
long minutes = seconds / 60;
long hours = minutes / 60;
long days = hours / 24;
/* Formats the above values into DD:HH:MM:SS. */
String time = String.format("%02d:", days) + String.format("%02d:", hours%24)
            + String.format("%02d:", minutes%60) + String.format("%02d", seconds%60);


textSize(24);

if (avgScore.size() > 1) {
  /* Checks to see if the avg. score from last generation is better than the generation before it. */
  if (avgScore.get(avgScore.size() - 2) < avgScore.get(avgScore.size() - 1)) {
    fill(0, 255, 0);
  }
  else {
    fill(255, 0, 0);
  }

  text("Avg fitness: " + avgScore.get(avgScore.size() - 1), 20, 640);
}

fill(0);
text("Generation: " + gen, 20, 680);
text("Number dead: " + pop.getNumberDead(), 20, 720);
text("Score of best: " + pop.players[0].level.score, 20, 760);
text("Framerate: " + frameRate, 20, 800);
text("Players rendered: " + playersRendered, 20, 840);
text("Time since start: " + time, 20, 880);
```

After running the program, this is what I see:

This shows that it is clearly working. I can change the number of players rendered currently by using the + and - keys as intended.

Log 5 – March 20th:

To save the current state of the program, I need to store all the values in the Population.players array. This can be done by utilising Processing's JSON capabilities. Processing has built-in libraries that allow me to save the current state of an object to a JSON file and then load it again in a future execution of the program.

To help with this, I have created a new function called save() in the NeuralNetwork class which returns a JSONObject, consisting of all of the weight matrices and dimensions for that NN. I have also created a new constructor in this class to easily load a NeuralNetwork object from a JSONObject when I load the players from a file.

This should allow me to save the current state of the program and then reload it once I run the program again. This is tested in test 3.

Prototype Three:

Testing:

Prototype 3 is primarily visual testing. No data is required specifically, other than the data naturally provided by the snake's learning process (i.e. the maximum and average scores of the snakes) and the data generated by the player's neural networks when the current state of the program is saved. There will be a lot of testing and experimenting with the parameters of the program, including the population size and neural network structure, which is heavily detailed in the program optimisation section.

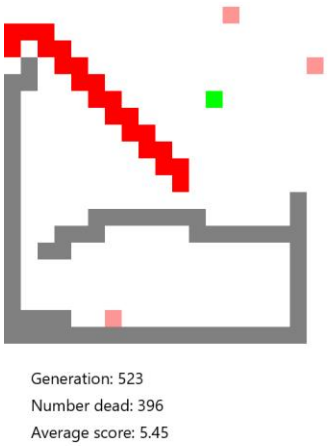
Test No.	Test Description	Expected Outcome	Actual Outcome
1	Testing NeuralNetwork.show() on the best player in Population.players.	This is not the finalised version of the method, but an initial test without bias nodes which will be added later. I should see all neurons in the NN as well as the weights between them.	Not as expected; see result 1.
2	Testing Graph.show() with the live data coming from the average and max score of previous generations.	x-axis and y-axis labels should render correctly, with no overlapping text. The line should correctly align with the y-axis (i.e. the maximum height of the line should match up with the height of the top value label).	As expected; see result 2.
3	Testing Population.save(), Population.load(), NeuralNetwork.save(), NeuralNetwork(JSONObject)	I should be able to run the program for a while and then close the program, and change the program to load the population from a file the next time the program is launched. This should then load all the progress from the previous run of the program and continue execution.	Not as expected; see result 3.

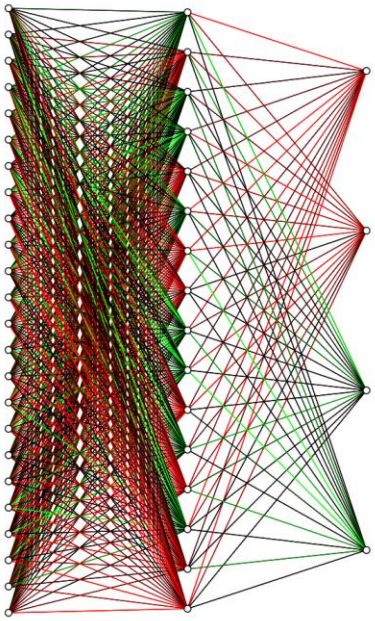
Test result 1:

This is the code I used to test this:

```
pop.players[0].nn.show(600, 0, 800, 950, 10);
```

This is in the Main.draw() method so it is executed every frame to draw the first player in the players array within the Population object 'pop'. The Population class sorts the Player array so that the first item in the array is the best-performing snake. This means that the neural network shown is the current best-performer. This is the result:





Prototype Three:

It's clear to see that the network is displaying as expected; the network structure is displaying correctly, with 24 input neurons, 16 hidden layer neurons and 4 output neurons, which is the structure I used for this test. Furthermore, you can clearly identify which weights are positive and which are negative by the strength of the colour of each weight.

However, I ran the program again with a more extreme network structure of 24 input neurons, 50 hidden layer neurons and 5 output neurons and this was the result:

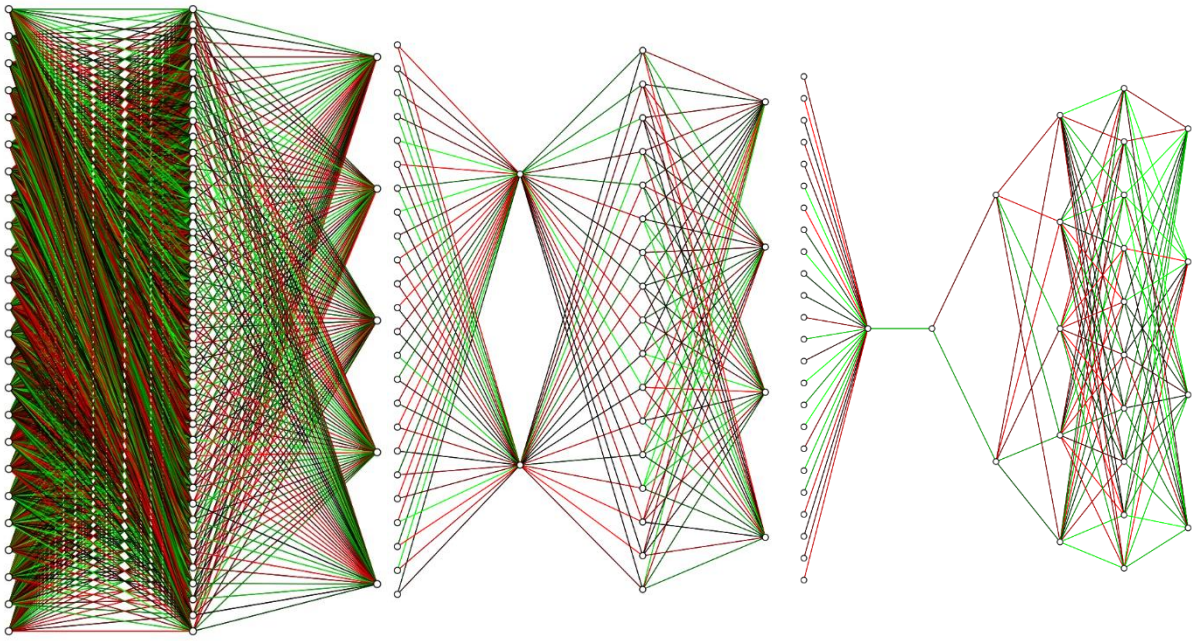
```
80          if (weightMatrices[i].data[i][j] > 0) {
81            stroke(0, 255 * weightMatrices[i].data
82          }
83          else {
84            stroke(255 * Math.abs(weightMatrices[i
```

ArrayIndexOutOfBoundsException: 25

```
ArrayIndexOutOfBoundsException: 25
ArrayIndexOutOfBoundsException: 25
```

I received this out of bounds error on the line highlighted (line 80). Upon inspection, it was obvious I accidentally referred to the weight matrix data using `weightMatrices[i].data[i][j]` rather than `weightMatrices.data[j][k]` as this is a triple nested loop. This would mean that the weight colours in the first test were not referring to the correct weights.

Below is the result after re-running with a range of different structures:



You can see it now works as expected. Even on unconventional networks, such as the rightmost network, the graph still rendered correctly.

Bias nodes are not being displayed in this diagram, as this method is unfinished, but as everything else is working correctly, this second test is successful.

Prototype Three:

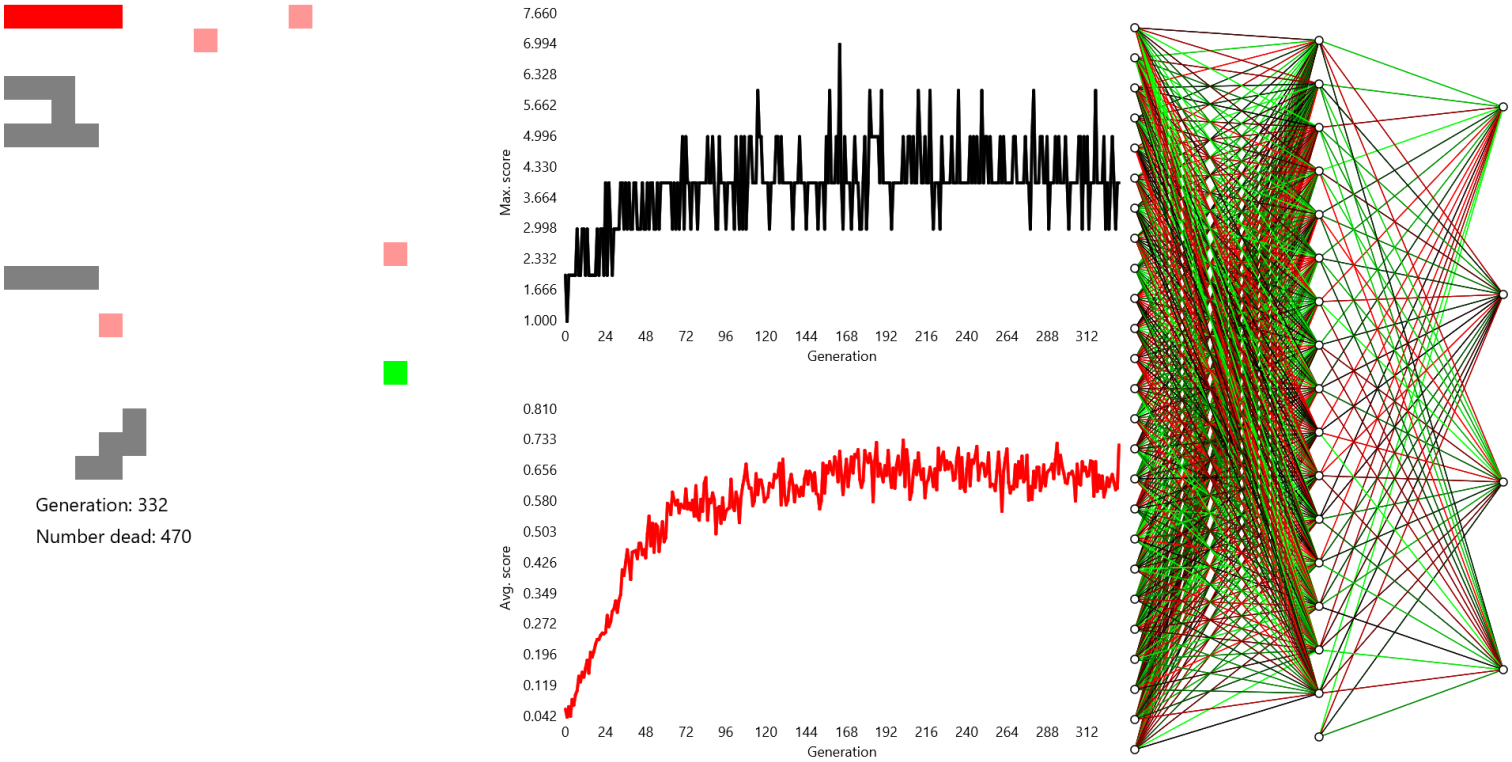
Test result 2:

Below is the code I am using to test this graph. I have setup both the maxScore graph and avgScore graph to test both at once.

```
maxScore = new Graph("Generation", "Max. score", color(0));
avgScore = new Graph("Generation", "Avg. score", color(255, 0, 0));

maxScore.show(700, 50, 700, 350);
avgScore.show(700, 550, 700, 350);
```

This is the result after running the program:



As you can see from the screenshot above, the graphs are clearly working. They don't seem to have any issues even when there are hundreds of lines being drawn to the screen. One issue I noticed at the beginning of the program is that when the value on the graph hasn't changed from the last generation, the line will not be drawn until a value is drawn to the graph that is higher. After looking at the code, I can see that the issue lies with working out the valueRange. If all data in the graph is the same, the valueRange will be 0, which causes issues. I added the following code to solve this:

```
if (valueRange == 0) {
  /* valueRange cannot be 0, so if it is, make it an arbitrarily small number. */
  valueRange = 0.000000001;
}
```

This issue was resolved after re-running (it is difficult to screenshot).

Another issue I can see is that in the y-axis of the maximum score graph, the values are all floating-point values even though all values in the maxScore graph should be integers. It would be more user-friendly and easier to read if this axis were also integers, but it remains functional as a graph as it is now. If I was to create a further prototype to polish the user interface, this would be addressed, but as it is a minor issue, it is satisfactory for this prototype.

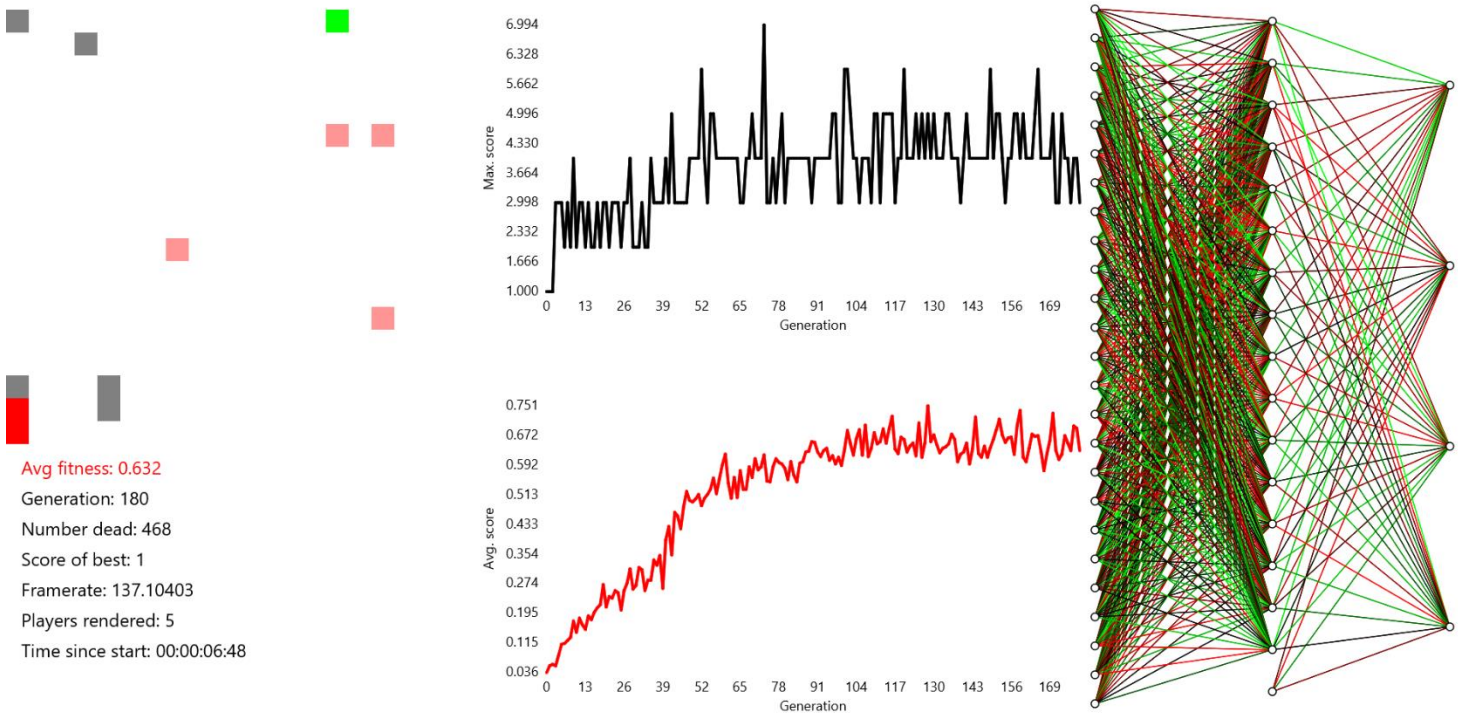
Overall, the graphs show as expected and, after fixing some minor issues, this test was successful.

James Ball

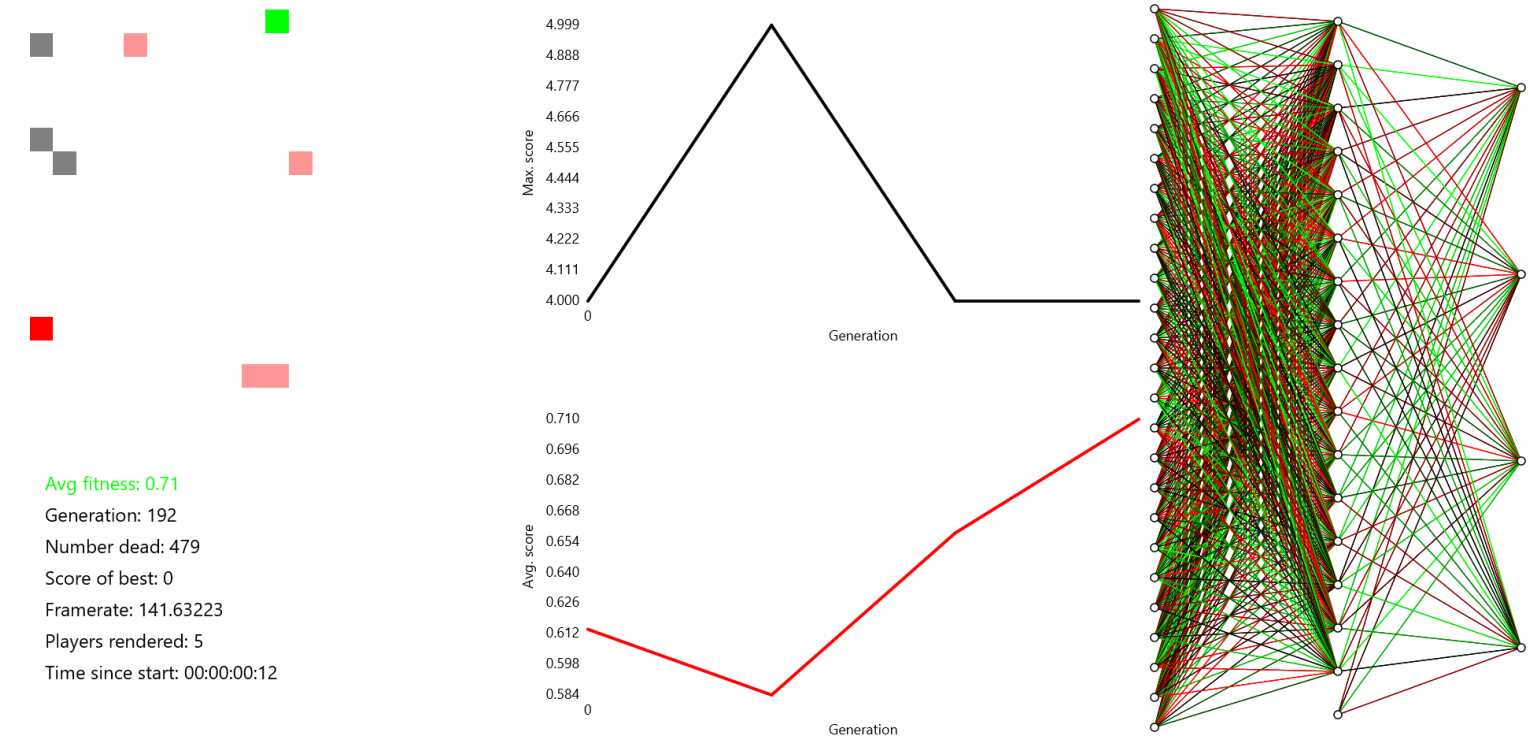
Prototype Three:

Test result 3:

After the end of each generation, the current state of the program should be saved to a file called program.json. I ran the program for a while to get a higher average score as you can see below:



The program reached an average of around 0.66. If I modify the code so that the program.json file is loaded the next time I run the program, I should immediately see a similar average. The generation number should also be kept. I ended the program on generation 188, so we should see it start here when I run it again. This is the result:



You can see that the average score is immediately around 0.66 as expected, and the generation number continues from 188 which it ended on last time. Although the files were correctly saved and loaded, there is an issue with how the graphs display, as they rely on the generation starting at 0, which it won't if the program is loaded from a file. To solve this, we can save and load the graph data as well as the neural networks so that they match up.

After adding code to save and load graph data, I re-ran the test and closed the program with an average score of around 0.55. These are the results after opening the program again:

```
271     /* Loads all graph data. */
272     for (int i = 0; i < maxScoreData.size(); i++) {
273       maxScore.add(maxScoreData.getFloat(Integer.toString(i)));
274     }
```

```
NullPointerException

NullPointerException
NullPointerException
```

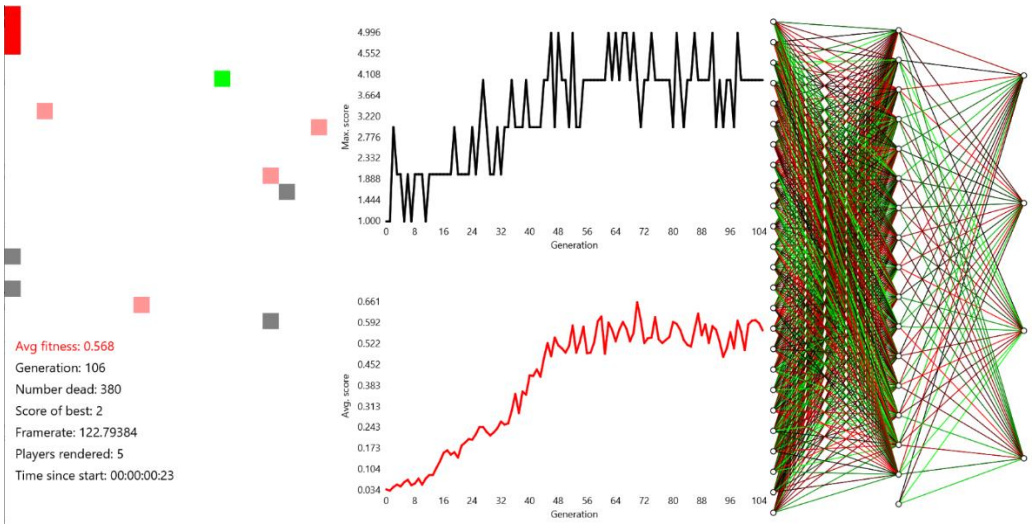
After receiving this error, I looked at the references for JSONObject and JSONArray provided by Processing, and realised maxScoreData and avgScoreData should be stored as JSONArrays rather than JSONObjects. I re-ran the program again with an average of around 0.55 again:

```
271     /* Loads all graph data. */
272     for (int i = 0; i < maxScoreData.size(); i++) {
273       maxScore.add(maxScoreData.getFloat(i));
274     }
```

```
NullPointerException

NullPointerException
NullPointerException
```

After debugging for a while by printing to the console and using breakpoints, I realised the JSONArray 'issue' wasn't causing this crash, it was because I hadn't yet initialised either of the graphs, so trying to add data to their lists was causing a NullPointerException. I re-ran it for a third time:



This time it was successful. All graph data was loaded correctly, and the generation labels were showing correctly. I can finally say that this feature is working as expected.

General outcome of testing and user feedback:

Initially, the tests seemed completely positive, but after I tested them with more obscure data, issues started to appear. For example, in test result 1, after testing with different neural network structures I discovered there was a big issue with the NeuralNetwork.show() class which was promptly fixed.

After re-testing and fixing these issues, the program has all the functionality I expected at this stage in development.

I spoke again with the initial focus group about the third prototype and they both agreed that it looks very similar to the initial sketch I made. The undergraduate student commented on the graphs and thought it would be extremely useful for when I try to optimise the program to improve the speed at which the snakes learn. The younger student thought the neural network graph was extremely interesting as it visualises what he was trying to learn about and gives an impression of how data passes through the network. Finally, the undergraduate student loved the amount of information provided underneath the snake grid, as you can see from the screenshot above in test result 3.

All extra functionality for this prototype is complete, but the next section details the steps I will make to try to optimise the program further.

Program optimisation:

All features are implemented as expected, as will be addressed in the upcoming evaluation, however there are some inefficiencies with the learning algorithm and some parameters that can be fine-tuned to improve performance. The parameters that need modifying include the following:

- populationSize – The greater the size of the population, the higher the likelihood of a snake learning something new, but also the slower the performance.
- networkStructure – I need to find an appropriate structure that isn't too complex that the snakes will take far too long to learn, but that isn't too simple that the snakes can never learn any advanced techniques.
- mutationRate – Increasing this will increase the likelihood of a better performing snake but increasing it too much will cause too many negative mutations and cause the population's average to suffer as a result.
- Activation function – The activation function (explained briefly in Analysis section under 'Neural Network') that I use currently is the sigmoid function (see Prototype 2 test result 5), but there are other activation functions to choose from which could potentially improve performance.

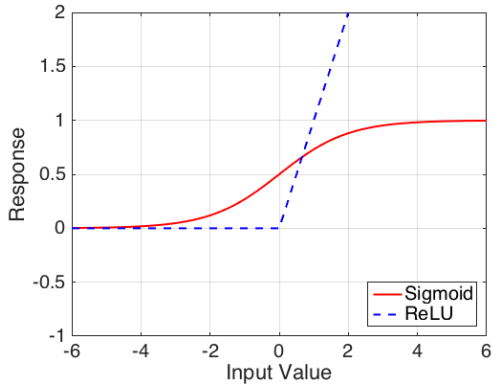
All these parameters will have optimal values where they increase the speed at which snakes learn without over or under-complicating the problem. This section tests different combinations of these parameters to try and find an optimal set of parameters.

The 'default' parameters I will use are a populationSize of 500, networkStructure of 24 input neurons, 16 hidden layer neurons and 4 output neurons, a mutationRate of 2% and the sigmoid activation function.

With each parameter test, I will run the program 5 times until generation 200 and then average the max average score achieved at generation 100 and 200 across all runs.

Prototype Three:

The alternative activation function I am using is called ReLu (Rectified Linear Unit). This checks the output of a layer in the NN and if it is positive, the output is not changed, but if it is negative, it is changed to 0 instead. You can see this in graph below, in which Sigmoid and ReLu are both plotted.



You can see that Sigmoid is a smooth function that has a maximum value of 1 and a minimum of 0. ReLu is quite different in that it is either 0 if the input value is less than 0, or the same as the input value if the input value is greater than 0.

ReLu is supposedly better at mimicking natural selection and supposedly performs better, so I will compare Sigmoid and ReLu below to see which works better for my program.

The results are collated below:

Population size	Network structure	Mutation rate	Activation function	Average score at generation 100	Average score at generation 200
500	24, 16, 4	0.02	Sigmoid	0.4586	0.5024
500	24, 16, 4	0.05	Sigmoid	1.5752	3.5014
500	24, 12, 4	0.05	Sigmoid	2.343	3.9526
500	24, 8, 4	0.05	Sigmoid	0.3608	0.4356
250	24, 12, 4	0.05	Sigmoid	0.399	0.6746
1000	24, 12, 4	0.05	Sigmoid	0.413	1.585
500	24, 12, 4	0.1	Sigmoid	2.2764	4.381
500	24, 16, 4	0.02	ReLu	2.7548	10.3622
500	24, 12, 4	0.02	ReLu	8.1848	16.0582
500	24, 10, 4	0.02	ReLu	7.9432	15.5724
500	24, 10, 4	0.05	ReLu	4.3178	8.8444
500	24, 8, 6, 4	0.02	ReLu	0.6464	1.9144
500	24, 12, 4	0.05	ReLu	3.6528	6.0844

After running these tests, it's clear that some of these changes make a big difference. Population size seemed adequate at 500 for all these tests so I kept it the same overall. Increasing the mutation rate when I used the sigmoid activation function increased the performance, but after changing the activation function to ReLu, there was clearly a huge improvement in the performance of the genetic algorithm. The first test using sigmoid had an average at 0.5024 at generation 200, compared to an average of 10.3622 at generation 200 when completing the same test with the ReLu activation function. This increase by a factor of 20 is clearly a huge improvement in performance.

I further improved upon this by modifying the network structure to 12 hidden neurons, rather than 16. This change increased the average after 200 generations by almost 6; another big improvement.

Optimising the program has led to approximately 32 times the performance after 200 generations compared to the initial parameters being used, which is very successful for this short and limited test. Performing more tests would lead to further optimised results, but for now the parameters I will use are as follows:

Population size of 500, network structure of 24 inputs, 12 hidden layers and 4 outputs, mutation rate of 2% and ReLu activation function, rather than sigmoid.

I have changed the default parameters in this prototype to reflect this.

Evaluation:

In this section, I will compare Prototype 3 against the success criteria set out in the Analysis section.

1. Graphs must be generalised to fit a variety of data and labels.

   The Graph class was implemented in a way that allows for customised x and y-axis labels for any range of data. In the case of this program, the maximum score and average score of each generation was used, but this could have easily been changed to another type of data by just changing the x and y-axis labels and adding different data. This allowed me to focus on the data and labels of the graph, rather than how the graph works, which I mentioned in this specific criterion. This criterion was clearly met.

2. Graphs must be simplified as the number of data items increase.

   I mentioned a system in which datapoints are removed as the complexity of the graph increases to save performance, but after testing evidenced in test result 3, the performance of the program is affected minimally, even at high generation numbers where there are a lot of datapoints. I have met this success criterion in other ways, however, including the fact that labels on the x-axis will simplify and are removed when there is not enough space left. This was not intended as a performance benefit, but serves a purpose, nonetheless.

   Therefore, this criterion has not been met, but with the reason that the performance was not significantly affected as the number of data items increase, so meeting it was not beneficial.

3. Neural network graphs should show all connections between nodes and layers and visualise their weights.

   This has clearly been met through the NeuralNetwork.show() method. This visualises the neurons in the network and all weights between neurons. I have also met the criteria listed by colouring the weights depending on the positivity or negativity of the weights. This can all be evidenced in development log 1.

4. The current state of the snake's neural network should be able to be saved and loaded so that a known-working snake can be loaded in future executions of the program.

   I have gone above and beyond this specific criterion, as I haven't only added functionality to save and load the best neural network, the whole program-state is saved and loaded using Population.save() and Population.load(). Not just the best neural network, but all currently running neural networks are saved to a file, along with all graph data for the program. This means that when I re-run the program, all progress moves over from the last execution. This is evidenced in test result 3.

   I mentioned that the neural networks should be saved by pressing a button, but I have opted instead for a system that saves the program-state at the end of each generation so make the process more automatic.

It should be clear that I have either met the criteria or explained why certain parts of it have not been met.

Overall User Feedback:

After fully completing and optimising prototype 3, I spoke with my focus group for the last time. Both students ran the program and allowed the snake to learn for a while before I stopped the program and then re-opened the program to test the saving and loading functionality. After this, I stopped the program again and asked some questions. Their responses are summarised below:

Questions:

Was it clear to see that the snakes were learning how to play the game?

The undergraduate student thought it was obvious there was a clear change in the snakes. He mentioned you can see the snakes early on not achieving anything, but as their learning progresses, they begin to play the game in a more humanoid manner. The younger student didn't add anything but agreed with this.

Were there any issues with visualising the snake games?

Both students agreed that the snake game is visualised correctly. The younger student thought it would be beneficial to make each snake have a unique colour when displaying multiple snakes so they can be more easily told apart.

Did the graphs and neural network diagram render correctly?

Both thought the graphs rendered correctly, but the undergraduate student thought it would be clearer to only show integers on the y-axis of the graphs (especially on the Max Score graph that only uses integer values).

Did you notice any performance issues?

The students agreed that the performance was consistent and there were no specific issues with it, but the undergraduate student mentioned performance could be greatly improved if I ran the program without a user interface and utilised multitasking. A separate program for this could be implemented in a further prototype – as detailed in the overall evaluation.

Could you see when the population of snakes make an innovation?

Before asking this, I clarified that an innovation is a mutation that greatly improves the score of the population after it is learned. The students both agreed that the graphs make it clear when an innovation occurs. They said that the average score and max score graph spike when the population of snakes learn something new.

Analysis of feedback:

Feedback was generally positive, with a few things that could be improved upon. Notable issues are the colours of the snakes being rendered when there are more than one on the grid, the graphs not displaying integer values on the y-axis and the potential performance increase of creating a standalone program. Looking forward in future prototypes, I would create a separate program that implements multitasking, along with improvements to the current program to address the issues mentioned.

Further reference to this feedback is made in the evaluation below.

Overall Evaluation:

Explanation of meeting success criteria has been discussed at the end of the appropriate prototype, but this section details the evaluation of the whole program.

Maintenance:

I have focused the design of classes in this project around future maintenance. As a result, using classes in this program is very easy as they are all clearly defined and self-explanatory to use. Maintenance can easily be carried out by utilising the parameters provided in the Main class, such as the mutationRate, networkStructure and populationSize. I have demonstrated simple maintenance of the program in the Program optimisation section of Prototype 3. The naming of classes and variables creates self-documenting code that when read by another programmer is easy to understand and doesn't require excessive commenting to explain what each variable does, for example.

If extra graphs were to be added, I have made it very simple to create a new Graph object in the Main class and then show it each frame. Data can be simply added to the graph and showing the graph again will reflect this addition.

The NeuralNetwork class has been designed to support any networkStructure specified, and NeuralNetwork.show() will correctly visualise this.

If changes to the size of the grid are to be changed, the gridX and gridY values can be changed and the change will immediately be reflected next time the program is run.

If extra functionality were to be added to any part of a class, there is a clear distinction between classes, making it easy to know which class to update. If you were to make a change to the genetic algorithm, you'd modify Population or if you were to change how neural networks work, you would clearly use the NeuralNetwork class.

These points highlight the ease-of-maintenance I have tried to achieve when designing classes.

Limitations:

The amount of detail I can put into graphing the average and maximum scores for each generation is one limitation. There is a significant computational requirement to display these graphs due to the potential complexity of the Graph.show() method. If I were to spend more time on this method, it could become very complicated and more computationally taxing for the added benefit of displaying data in a more user-friendly way. Graphs are not the primary focus for this project, which is why they have not been fully-developed and simply meet the program's requirements and nothing more. The user feedback highlights some of the issues with the graphs, including the fact that integer values are not displayed on the y-axis, which would make it easier to read.

The performance of the genetic algorithm is another limitation. You could spend an endless amount of time finding the perfect parameters for snakes to learn optimally, but there is not enough time during development of this project to devote lots of time to optimising the program. I have a Program optimisation section, but this could easily be more comprehensive with the addition of more parameters, more tests and tests to a higher number of generations.

Performance of the program in general is a hugely limiting factor. The Processing framework/IDE does not have suitable libraries for concurrent processing, making it hard to create a program that makes effective use of multiple cores on a computer. This is because Processing is a subset of Java and doesn't implement all the threading libraries Java uses. When running the program at the

maximum speed my computer can handle, the program only uses a maximum of 20% of the CPU as my computer has a high number of cores. If I could re-write this with a different library that supports concurrency well, performance would benefit hugely. This re-write is mentioned in the Further Development section below.

User feedback for further prototypes:

This section details all the feedback I would use from my focus group if I were to create further prototypes.

The undergraduate student thought graphs would look clearer as having integers only on the y-axis, so I would implement this feature in the graph.show() method, as mentioned in the 'Further Development' section. I would also make the currently rendered snakes more distinguishable by making their colour unique, as suggested by the younger student. This would make the user interface more user-friendly and easier to understand.

Finally, the undergraduate student mentioned performance would be increased if I created a separate program with no user interface. This would be implemented in a further prototype. The details of this standalone program are mentioned below:

Further Development:

If I were to develop a fourth prototype, I would move all the code from the Processing framework/IDE into a purely Java program so that I could make use of the concurrent processing functionality that native Java allows. array.parallelStream().forEach() is a function that Java allows which allows me to parallelize execution of an array of objects. I could perform this on Population.players when calling Player.update(). This would utilise most of my processor's performance, rather than the limited amount utilised in prototype 3.

This standalone program would still support saving all graphing data and neural network data to a file that can then be read by the program created in prototype 3. This would allow the snakes to be trained in a program without any user interface and that supports parallel processing and then loaded into a program with a visual interface. This speeds up the learning process but still allows the snake's progress to be visualised.

As well as dealing with the performance of the program in general, this prototype would mitigate the effects of having a genetic algorithm that performs poorly. This is because the program will be running dramatically faster, so it is less important that the genetic algorithm is not optimal. Despite this, it would still be beneficial to run many more tests on the various parameters of the program in this prototype to reduce the effect of this limitation.

A fifth prototype could address the limitations relating to graphing. I would like to increase the capabilities of the Graph class to support correct rendering of the maxScore graph as all datapoints on this graph are integer values. Currently, the graph's y-axis labels are all floating-point values, although the maxScore graph data consists of only integers. This user-interface change would make the graphs more readable. This is reflective of the feedback provided in the overall user feedback.