Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC5114 Neural Networks and Genetic Programming

Homework N°3

NeuroEvolution of Augmenting Topologies Algorithm, Applied to Snake

Lecture Professor: Alexandre Bergel
Assistant Professor: Juan-Pablo Silva
Assistants: Alonso Reyes Feris
Gabriel Chandía G.
Alumni: Diego Ortego
RUT: 19.671.411-1
Github: Gedoix
Project: link to github
Date: January 5, 2019

1 Introduction

This homework's goal is to code and test an extensible object-oriented implementation of the `N.E.A.T.` (`NeuroEvolution of AugmentingTopologies`) algorithm as it's originally proposed by Kenneth O. Stanley and Risto Miikkulainen on their paper Evolving Neural Networks through Augmenting Topologies, and to test this implementation as a learning agent that plays the well-known video-game Snake.

The specifics on the benefits on the `N.E.A.T.` algorithm for learning agents in most fields will only be quickly glossed over in this report, as a lengthy full explanation on them can be found on the fore-mentioned paper proposing the method.

The project was developed using Jetbrain's Pycharm IDE and built around instructions and guidelines given by the course's lectures.

The project can be found in this github repository.

The necessary packages, installation instructions, and other information regarding tests can be found in the project's `README.md` file.

All files concerning this document are inside the project's `src/neuroevolution_algorithms` and `src/snake` packages, and recorded results can be found in the `plots/snake` directory.

2 Methods

2.1 Problems to Solve

2.1.1 N.E.A.T. Implementation

The first problem to address is the implementation of the algorithm.

Although the paper proposing it goes to great lengths as to convince readers of it's efficiency and effectiveness, it doesn't include any source code from which to base an implementation off of directly.

Not only that, but the guidelines and explanations it offers as to how it could be done are difficult to put in practice, even with a good data structure choice.

Since this is the case, part of the challenge of the project was to come to understand the algorithm well enough to put it's principles, goals, and the author's guides, together into a working class structure capable of learning.

The algorithm is a form of genetic evolution-inspired optimizer, where the figure being optimized is the fitness value of a classifier neural network.

To do this, it creates a population of small networks and allows them to slowly grow in complexity and adjust to the problem's requirements.

The algorithm also includes a way of organizing the network's structures that permits crossover as a method of generation of new population members without the need to analyze compatibility between sub-graphs, a known to be complex problem, by leveraging on the historic information of how the networks evolve over time.

To do this it also suggests ways to separate the population into species, limiting their amount of competitors and potential crossover candidates to only those of similar genetic history.

The program should do the following:

- Generate a population of basic neural networks that have no hidden layers, composed of only input and output nodes.
- This population then is evaluated by an abstract fitness function, so it can recognize the quality of each member's results.
- The fitness function to use depends on the problem at hand, so it's important to keep it open for modifications.
- Then, population members with low fitness values are removed from the population, and a new population is generated from the survivors, allowing some exceptional population members to survive unscathed and generating a small amount of new ones from scratch too.

- This new population is composed of new nodes created through crossover between their parents, as well as new nodes created from copies of the old ones, which have been mutated do allow for genome variety to arise.
- The new population then has it's fitness values evaluated.
- It's then categorized into species, where two individuals belong to the same species if the distance from each to the species's representative is smaller than some threshold.
- The distance between individuals is calculated as a function of the amount of synapses the networks have in common, the amount they don't and the average difference in the weight values they use.
- Finally a shared fitness value for each species is calculated, from the fitnesses of each of it's members. This number decreases with the size of a species, and increases with the individual fitnesses of each member.
- The program loops, starting again from the 5th item in this list, until a target fitness is achieved by any of it's agents.

As previously stated, the advantages of the algorithm, if not obvious, and a throughout analysis of the concepts that inspired it and how it works can be found in it's original proposition.

2.1.2 Test as a Snake-playing Actor

The second problem to address is the implementation and execution of a game of Snake, and finding a way for the algorithm to serve as a learning player actor for the game.

The goal is for the algorithm to be able to find one of the possible configurations of it's network that will allow it to reach a score of 40 eaten fruits in a 11×11 board of the snake game.

The inputs and outputs the snake possesses for taking it's decisions are the following:

- Three normalized (using the modified hyperbolic tangent function in figure 1) 0 to 1 values representing the distance from the snake's head to the walls to it's left, front and right sides, considering it's movement direction at the time of calculation.
- Three equally normalized 0 to 1 values representing the distance from the snake's head to any fruits to it's left, front and right sides. Although the game is configured to only allow one fruit on board at any given time, the system allows for more. If no fruits are found, the normalization function is fed the biggest 64 bit integer available.
- Three equally normalized 0 to 1 values representing the distance from the snake's head to any body part it has to it's left, front and right sides. If no body parts are found, the normalization function is fed the biggest 64 bit integer available.
- Another equally normalized value representing the snake's current length.

```
31
32  def normalize_distances(x: float) -> float:
33      import math
34      return math.tanh(x)
35
36
37  def normalize_score(x: int) -> float:
38      import math
39      return math.tanh(x/100.0)
40
```

Figure 1: Hyperbolic tangent normalization functions for the network's inputs.

It's easy to see that simple decision-tree-like controller can easily be written by a human to keep the snake away from walls and it's own tail while eating the most fruit available.

This doesn't mean, though, that the algorithm won't have trouble finding it's own solution to the problem.

2.2 Solution Scheme

Going after the goal of building an implementation of the algorithm that could manage the snake game, stay extensible and well encapsulated, the scheme is follows:

- The neat algorithm needs a class of it's own.
- The constructor, or builder, method of said class must allow it to receive arbitrary specifications on how to generate populations, evaluate them, and reproduce them.
- The algorithm should be able to answer basic performance-related questions about itself during execution, so that these values can be visualized and compared.
- For testing and reproducibility purposes the algorithm's results should also be completely determined by it's pseudo-random number generator's seed.

The initialization of the algorithm should setup a starting population of empty networks, each carrying it's own differently seeded random number generator.

Lets call the class through the variable name `neat` for ease of communication purposes, initialized with the obligatory parameters:

- `input_amount`, the length of the `float` input list the networks should be able to compute from.
- `output_amount`, the length of the `bool` output list the networks should be able to produce.
- `fitness_function`, function the algorithm will use for evaluating networks in each cycle.

And the optional parameters:

- `population_size` amount of networks to keep as a population.
- `fitness_stagnancy_cap` relative in-population fitness percentage separating deletable and not deletable networks.
- `large_species_size` size threshold of a large species. The algorithm discriminates small species as non-deletable until they grow enough.
- `mutation_change_weights_chance` percent probability of happening on every new network birth for a mutation that changes the network's weights.
- `mutation_add_synapse_chance` percent probability of happening on every new network birth for a mutation that adds a synapse to the network.
- `mutation_add_neuron_chance` percent probability of happening on every new network birth for a mutation that adds a new neuron to the network.
- `no_crossover_chance` percent probability of happening on every new network birth for the child to be a copy of one of the parents, with no crossover whatsoever.

- `inter_species_mating_chance` percent probability of happening on every new network birth for the child to be a the crossover product of two different species.
- `excess_distance_coefficient` coefficient weighing the difference in synapse amount between two networks, used when calculating the distance between them (for species separation).
- `disjoint_distance_coefficient` coefficient weighing the amount of different synapses between two networks, used when calculating the distance between them.
- `weights_distance_coefficient` coefficient weighing the average weight difference between two networks, used when calculating the distance between them.
- `species_distance_cap` distance threshold after which two networks are not in the same species.
- `seed` Seed of the pseudo-random generators. Determines the whole of the result.

The method `neat.advance_generation()` executes a single iteration of the algorithm, running the methods `neat.__remove_useless_species()`, `neat.__reproduce_population()`, `neat.__calculate_fitnesses()`, `neat.__sort_population_by_fitness()`, `neat.__separate_species()` and lastly `neat.__calculate_shared_fitnesses()`, in that order.

Removing useless species implies taking the species information from the last generation and blocking big species that underachieve from reproducing. This is done by measuring their sizes, and the relative fitness of the species's best member in the population.

Reproduction means re-using the information of the species that were allowed to survive to generate new individuals.

This particular algorithm introduces parameters for the chance of any given offspring to be affected by a number of mutations, like random modification of synapse weights, adding a new synapse, of adding a new neuron.

The fittest individuals in every species, as long as the species is big, is copied directly into the next generation with no mutations.

Calculating fitnesses means calling the `fitness_function` parameter for the collection of all individuals in the population, and recording their individual fitnesses.

Species separation is done by measuring the distance between each network and a set of species representative networks. If the distance is smaller than the threshold parameter the individual network is added to the species.

If no matching species representative is found, the individual itself becomes a species representative for a new species.

Finally, When calculating shared fitnesses, the algorithm scans all species and divides the sum of their individual fitnesses by the size of the species.

A global shared fitness is calculated as the sum of all species's shared fitnesses.

This then allows the algorithm to remember the importance of each in comparison with the whole.

This process repeats until the user evaluates the resulting best individual, accessible through the `neat.get_best_network_details()` method or indirectly thorough the `neat.get_best_fitness()` method, as good enough.

This condition needs to be set externally.

So a loop can take the form:

```
1 neat.advance_generation()
2 i = 0
3 while neat.get_best_fitness() <= TARGET_FITNESS:
4     print("The current best fitness in the population is = " + str(neat.
          get_best_fitness()))
5     neat.advance_generation()
6     i += 1
7 print("The maximum fitness was found after " + str(i) + " iterations")
```

2.3 Implementation

2.3.1 N.E.A.T.

Inside the `src/neuroevolution_algorithms` package, in the `neat_network.py` file 6 `Python` classes, used for building the N.E.A.T. network structure, were implemented:

```
class GeneticNeuron:

    __calculations: int

    # Constructor

    def __init__(self):...

    # Calculations counter

    def set_calculations(self, calculations: int) -> None:...

    def get_calculations(self) -> int:...

    # Main behaviour

    def calculate(self) -> None:...

    @staticmethod
    def modified_sigmoid(x: float) -> float:...

    def add_input(self, synapse: "GeneticSynapse") -> None:...

    def add_output(self, synapse: "GeneticSynapse") -> None:...

    def remove_input(self, synapse: "GeneticSynapse") -> None:...

    def remove_output(self, synapse: "GeneticSynapse") -> None:...


class InputNeuron(GeneticNeuron):...


class HiddenNeuron(GeneticNeuron):...


class OutputNeuron(GeneticNeuron):...

```

```
39
40 class GeneticSynapse:
41
42     __availability: bool
43     __weighted_value: float
44     __end_neuron: GeneticNeuron
45     __weight: float
46     __start_neuron: GeneticNeuron
47
48     # Constructor
49
50     def __init__(self, start_neuron: GeneticNeuron, weight: float, end_neuron:
           GeneticNeuron):...
51
52     # Relay system
53
54     def relay(self, result: float) -> None:...
55
56     def fetch(self) -> float:...
57
58     # Availability management
59
60     def enable(self) -> None:...
61
62     def disable(self) -> None:...
63
64     def is_available(self) -> bool:...
65
66     # SETTERS
67
68     def set_start(self, start_neuron: GeneticNeuron) -> None:...
69
70     def set_weight(self, weight: float) -> None:...
71
72     def set_end(self, end_neuron: GeneticNeuron) -> None:...
73
74     # GETTERS
75
76     def get_start(self) -> GeneticNeuron:...
77
78     def get_weight(self) -> float:...
79
80     def get_end(self) -> GeneticNeuron:...
81
82
```

```
83  class Network:
84
85      # All Neurons in the network
86      __input_neurons: List[InputNeuron] = []
87      __hidden_neurons: List[HiddenNeuron] = []
88      __output_neurons: List[OutputNeuron] = []
89      __calculations: int
90
91      # All synapses in the network
92      __synapses: List[GeneticSynapse] = []
93
94      # Abstract representation of a synapse, using indexes of neurons in the network's
            lists as ids
95      __symbolic_synapses: List[Tuple[int, bool, int, bool]] = []
96
97      # Pseudo-random number generator
98      __generator: random.Random = random.Random()
99
100     # Fitness of the network as a classifier
101     __fitness: float = 0.0
102
103     # Shared fitness of the network, altered by it's species
104     __shared_fitness: float = 0.0
105
106     # CONSTRUCTOR
107
108     def __init__(self, input_amount: int, output_amount: int, seed: int = None):...
109
110     def __initialize_synapses(self):...
111
112     # FACTORY METHOD
113
114     @staticmethod
115     def new(input_amount: int, output_amount: int, seed: int = None) -> "Network":...
116
117     # CLONING
118
119     def clone(self) -> "Network":...
120
121     # MAIN EXECUTION
122
123     def calculate(self, inputs: List[Union[int, float]]) -> List[bool]:...
124
125     # MUTATIONS
126
```

```
127     def mutation_change_weights(self) -> None:...
128     def mutation_add_synapse(self) -> None:...
129 
130     def mutation_add_neuron(self) -> None:...
131 
132     # CROSSOVER
133 
134     def crossover(self, other: "Network", share_disjoints: bool) -> None:...
135 
136     # COMPARATOR
137 
138     @staticmethod
139     def compare(network_1: "Network", network_2: "Network") -> Tuple[float, float,
            float]:...
140 
141     # SETTERS
142 
143     def set_fitness(self, fitness: float) -> None:...
144 
145     def set_shared_fitness(self, shared_fitness: float) -> None:...
146 
147     # GETTERS
148 
149     def get_fitness(self) -> float:...
150 
151     def get_shared_fitness(self) -> float:...
152 
153     def get_io_signature(self) -> Tuple[int, int]:...
154 
155     def get_innovation(self) -> int:...
156 
157     def get_calculations(self) -> int:...
158 
159     # PROBABILITY MANAGERS
160 
161     def choose_with_probability(self, probability_percentage: float) -> bool:...
162 
163     def get_random_float(self) -> float:...
164 
165     def get_full_details(self) -> List[Tuple[int, bool, int, bool, float, bool]]:...
```

Inside the `src/neuroevolution_algorithms` package, in the `neat_network.py` file, a `Python` class, used for the N.E.A.T. algorithm, was implemented as follows:

```
1  class Neat:
2      __input_amount: int
3      __output_amount: int
4
5      __fitness_function: Callable[[List[Network]], List[Union[float, int]]]
6
7      __population_size: int
8      __fitness_stagnancy_cap: int
9      __large_species_size: int
10
11     __mutation_change_weights_chance: float
12     __mutation_add_synapse_chance: float
13     __mutation_add_neuron_chance: float
14
15     __no_crossover_chance: float
16     __inter_species_mating_chance: float
17
18     __excess_distance_coefficient: float
19     __disjoint_distance_coefficient: float
20     __weights_distance_coefficient: float
21
22     __species_distance_cap: float
23
24     __starting_seed: Optional[int]
25     __seed: Optional[int]
26     __generator: Random
27
28     __generation: int
29
30     __population: List[Network]
31     __symbolic_species: List[List[int]]
32
33     __shared_fitness_sums: List[float]
34     __total_shared_fitness: float
35
36     def __init__(self, input_amount: int, output_amount: int,
37                  fitness_function: Callable[[List[Network]], List[Union[float, int]]],
38                  population_size: int = 150, fitness_stagnancy_cap: int = 15,
                        large_species_size: int = 5,
39                  mutation_change_weights_chance: float = 80.0,
                        mutation_add_synapse_chance: float = 5.0,
40                  mutation_add_neuron_chance: float = 3.0, no_crossover_chance: float =
```

```
                    25.0,
41                  inter_species_mating_chance: float = 0.1, excess_distance_coefficient:
                        float = 1.0,
42                  disjoint_distance_coefficient: float = 1.0,
                        weights_distance_coefficient: float = 0.4,
43                  species_distance_cap: float = 3.0, seed: int = None):...
44
45      def advance_generation(self) -> None:...
46
47      def __remove_useless_species(self) -> None:...
48
49      def __reproduce_population(self) -> None:...
50
51      def __calculate_fitnesses(self) -> None:...
52
53      def __sort_population_by_fitness(self) -> None:...
54
55      def __separate_species(self) -> None:....
56
57      def __calculate_shared_fitnesses(self) -> None:...
58
59      @staticmethod
60      def __quicksort_by_fitness(population: List[Network]) -> List[Network]:...
61
62      def get_population(self) -> List[Network]:...
63
64      def get_fitnesses(self) -> List[float]:...
65
66      def get_best_fitness(self) -> float:...
67
68      def get_maximum_innovation(self) -> int:...
69
70      def get_shared_fitness_sums(self) -> List[float]:...
71
72      def get_total_shared_fitness(self) -> float:...
73
74      def get_best_network_details(self) -> List[Tuple[int, bool, int, bool, float, bool
            ]]:...
75
76      @staticmethod
77      def builder(input_amount: int, output_amount: int,
78                  fitness_function: Callable[[List[Network]], List[Union[float, int]]],
79                  seed: int = None) -> "NeatBuilder":...
80
81      class NeatBuilder:...
```

This code is fully documented within it's package, so only some small examples of usage are included here

An example of a construction for this class is:

```
neat = Neat.builder(input_amount=10, output_amount=3, fitness_function=self.
    fitness_function, seed=self.seed).\
            set_population_size(2000).\
            set_species_distance_cap(0.25).\
            set_mutation_add_neuron_chance(1.0).\
            set_mutation_add_synapse_chance(2.0).\
            set_mutation_change_weights_chance(85.0).\
            build()
```

Which creates a `neat` instance with the specified attributes, where `input_amount`, `output_amount` and `fitness_function` are obligatory parameters for the algorithm's native constructor.

An example of a learning loop for the algorithm is then the same as it was decided in the solution scheme:

```
neat.advance_generation()
i = 0
while neat.get_best_fitness() <= TARGET_FITNESS:
    print("The current best fitness in the population is = " + str(neat.
        get_best_fitness()))
    neat.advance_generation()
    i += 1
print("The maximum fitness was found after " + str(i) + " iterations")
```

2.3.2 Snake Game

Inside the `src/snake` package, in the `snake_game.py` file, the snake game was implemented using `PyGame` as the engine:

When executing the file, it runs a demonstration of the game, where the decision of whether to turn left, right or continue forward is asked to the user through the terminal.

The classes are:

- `AI` has a `choose(..)` method, which receives all inputs an AI learning to play the game receives, as stated in section 2.1.2.
- `Snake` is literally the snake actor for the game, keeps a list of the positions of it's body sections and has a `.step(fruit,score)` method. This method moves the snake according to the rules of the game after asking an instance of `AI` to decide on a direction.
- `Game` is the game instance. It has a `simulate(snakes,seed)` method that can simultaneously simulate snake games for the `Snake` objects it receives and return the iterations and scores they achieved. It also has a `show(snake,seed)` method, which can show a single game of snake on screen, as seen in section 6.

2.3.3 Testing N.E.A.T.

Inside the `src/snake` package, in the `NEAT_experiment.py` file, the testing of N.E.A.T. is achieved by creating a subclass of `AI` and overriding the `choose` method, this class is `ExperimentAI`.

This subclass is then used for constructing a `fitness_function` that can call `Game.simulate(...)` and use the scores and iteration amounts to create a fitness metric.

This function's code is as follows:

```
1     def fitness_function(self, population: List[Network]) -> List[Union[float, int]]:
2         # The seed changes
3         self.last_used_seed += 1
4
5         # Snakes are re-generated
6         snakes = []
7         for n in population:
8             snakes.append(Snake(11, Experiment.ExperimentAI(n)))
9
10        # Metrics are calculated
11        scores, times = self.snake_game.simulate(snakes, self.last_used_seed)
12
13        # The fitnesses are calculated
14        fitnesses = []
```

```
15          for i in range(len(scores)):
16              f = scores[i]*(1.0 + 1.0/float(times[i]))
17              fitnesses.append(f)
18
19          return fitnesses
```

And is declared inside the `Experiment` class.

`Experiment`'s `main()` method contains N.E.A.T.'s testing and can be run by executing this file.

It first generates an instance of `neat`, and records it's maximum fitness while allowing the population to evolve, stopping when the game's simulation returns a score of 35 or more.

The network that achieved the goal is saved into a `.txt` file, for future usage.

A graphical simulation of how the AI played can be played.

Finally a plot for the maximum fitnesses of each generation is saved.

The parameters used for this test were found after many trial and error attempts, this is because the parameters suggested in N.E.A.T.'s original paper are ill suited for this problem, and a bigger population with a smaller chance of mutation per individual found solutions with a considerably smaller time cost, and in a more stable way.

Again, more information can be found in the file's documentation.

3 Results

A `.git` recording of the solution's final game and the plot result of the test's fitness over time can be found in section 6 of the document.

An additional `.pdf` file containing everything printed by the program can be found in `plots/snake/standard_output.pdf`

Some screen-shot images were taken, and are also included in the document.

It can be observed in the best network's final simulation that it effectively learned tactics on how to avoid it's own tail, and achieved the limits of what's possible given that the AI can't know the full complexity of the tail it avoids.

It achieves an algorithm for getting high scores in the game that doesn't require any more information, nor for the AI to have any internal memory of it's past decisions and measurements.

The amount of generations needed for this is of 92.

The fitness versus generation graph 7 also shows that the algorithm finds solutions in a rather unstable manner.

The terminal output of the test shows that many species of solutions appeared during execution, and the number of species fluctuated throughout it, getting as high as 17 nearing the result.

4 Discussion

The results of the testing, combined with the additional information that parameters needed to be tinkered with a lot before obtaining a consistent solution, suggests that the algorithm, despite it's usefulness for generating neural network classifiers from scratch, and the fact that it leverages some genetic algorithm's advantages, like a high chance of exiting local optima, is still very unstable.

The parameters need to be set beforehand to fit the problem at hand, or otherwise the networks either don't ever escape small synapse count local optima, or grow indefinitely, wasting population space with too many useless species.

It's also easy to miss a good distance to set for species discrimination, provoking potential executions of N.E.A.T. that handle a single large population, bypassing the benefits of specialization.

5 Conclusions

From this implementation and experiment it can be concluded that:

- The N.E.A.T. algorithm is very hard to implement, even when being a general purpose solution to many classification problems. The time invested could be used better working into either implementing a smaller general purpose learning algorithm and testing it manually, or implementing a larger one that isn't as unstable.
- Despite it being difficult, there may be ways of automating the optimization of parameters for the algorithm , a game-changer way of making it accessible.
- The algorithm is still apparently $O(e^N)$ time-wise, which means that analytic solutions should be preferred when available, but since the complexity of the solutions the algorithm finds, in theory, scales to fit the problem it can also solve an enormous amount of categorization problems without the need to hard-code neural network structures, or study their theoretical advantages over each other.
- When no analytic solution is available, or the analytic approach is exponential too, and of higher order, executing this algorithm either to solve or measure the complexity of a problem seems like a good idea.

6 Figures

6.1 Snake

6.1.1 Snake Simulation

The gif of the AI's final run can be found over at GIPHY.

It's also available through github.

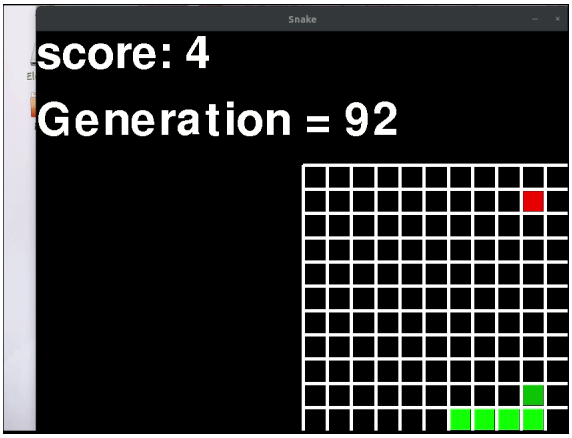

Figure 2: Screenshot taken from the gif presented above, at score 4.

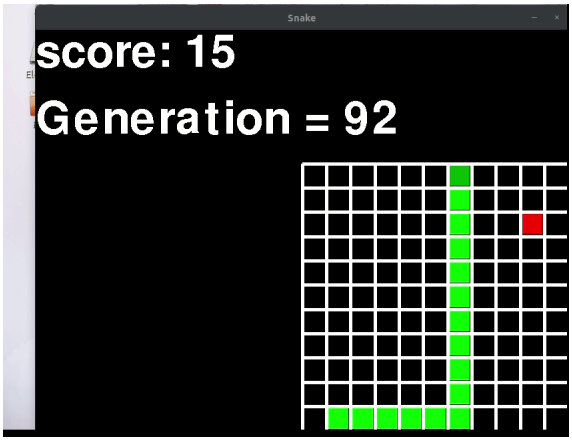

Figure 3: Screenshot taken from the gif presented above, at score 15.

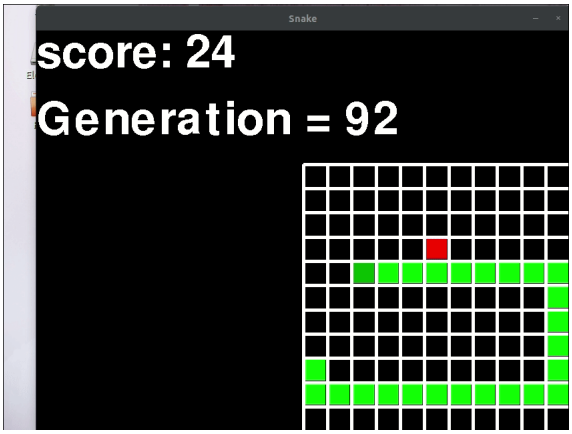

Figure 4: Screenshot taken from the gif presented above, at score 24.

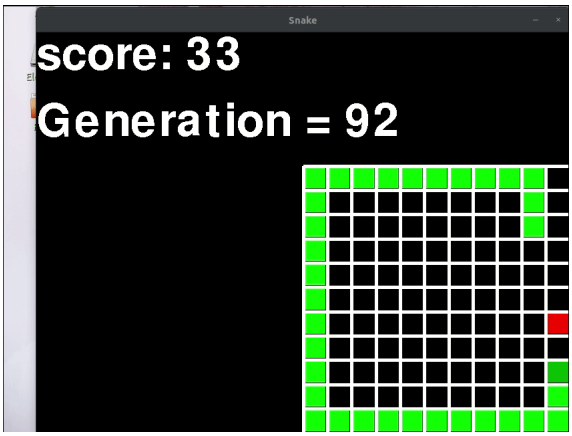

Figure 5: Screenshot taken from the gif presented above, at score 33.

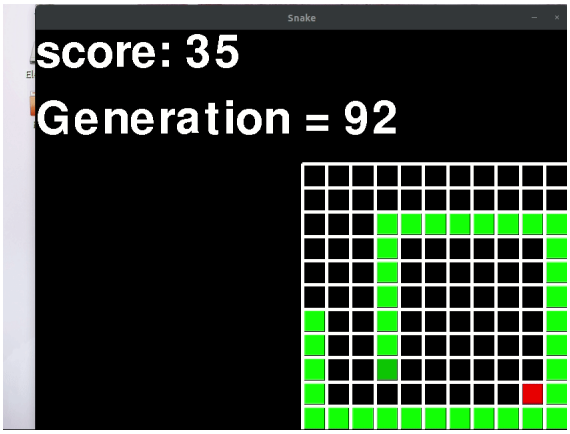

Figure 6: Screenshot taken from the gif presented above, right before the end of execution.

6.1.2 Fitness Plot

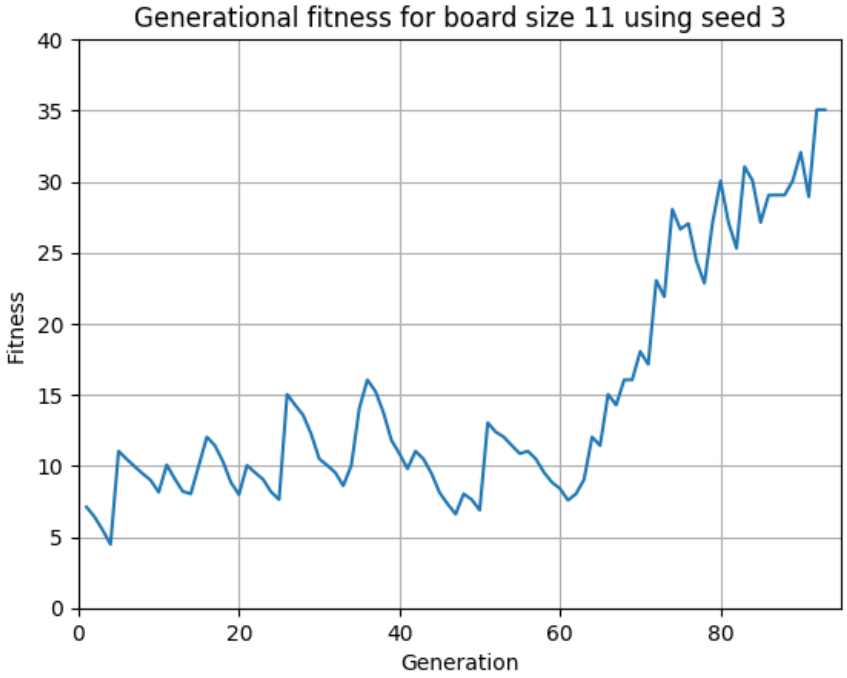

Figure 7: Fitness plot of the Snake AI for a board of side 11, and a fitness goal of 35