EVAC Assessment 1

Exam No: Y3868718

1 Methods

1.1 Changes to Provided Code

While the snake game discussed in this report is functionally the same as the initially supplied code provided with the assessment, some large changes have been made to the structure. Most notably, the snake game and neural network has been split into multiple separate scripts for ease of development. Details on all provided files can be found in appendix A.

1.2 Chosen Evolutionary Method

A Genetic Algorithm (GA) was used to evolve weights in a Neural Network (NN). Input nodes to this neural network were the snake sensing functions, with four output nodes representing snake direction: left, right, up and down. Heaton [1] discusses that most problems to not require more than one hidden layer, therefore a single hidden layer was used for the neural network with the number of hidden layer nodes corresponding to the sum of the input and output layer nodes. Using a single hidden layer has a benefit of reducing overfitting compared to a multiple hidden layer network, along with decreasing time taken to train the network. A softmax function is used at the output layer to obtain the direction the snake should take, where left=0, right=1, up=2 and down=3.

1.3 Development of Different Snakes

Four differing snakes were developed for the snake game and will be referred to in this report as:

1. `local-standard`
2. `space_aware-standard`
3. `local-n+n`
4. `space_aware-n+n`

Two directions of investigation were performed on the snake game, where each name is comprised of these two components separated by a hyphen. The first direction investigated the effect of differing collision sensing functions (`local` against `space-aware`) while the second investigated changing the generation selection method (`standard` against `n+n`).

1.4 Sensing Functions

Sensing functions were created for the snake to use to learn about its environment, these can be classified into two groups; food sensing functions and collision sensing functions. All sensing assumes grid format of `[y,x]` and directions are relative to the grid, not the current direction of the snake.

1.4.1 Food Sensing

Food sensing was constant throughout all developed snakes. Multiple methods of food sensing were explored during development including passing the exact position of the food, calculating the manhattan distance between food and snake, and sensing the quadrant of the grid the food was in relative to the head of the snake. The chosen method instead simply checks if food is in the direction specified, for left and right the x axis of the snake is checked while up and down checks the right axis. The sensing function returns a boolean value; True if food is present and False if not.

If the Snake is directly in line with food in that direction, both directions on the axis will return False. For example, a snake head at position [4,4] and food at [6,6] would return True from `downFood` and `rightFood`, and False from `upFood` and `leftFood`. A snake head at position [4,4] and food at [6,4] would return True from `downFood` and False from all other sense directions.

1.4.2 Collision Sensing

Two methods of collision sensing were developed, the first method of collision sensing is referred to as `local` and was used by both `local-standard` and `local-n+n`. Each direction has two collision sensing functions that both check based on a given position; one used to sense if the position is within a wall, and the other used to sense if the position is within the snake body. Both of these sensing functions return True if a collision is detected, False otherwise. These positions are given for each direction by calculating the position of the snake head if the snake moved in the given direction. For example, if the snake head is at position [4,4], the position given to `senseWall` and `senseTail` would be [4,3] when `leftWall` and `leftTail` are called.

These wall and tail functions could be combined into a single function to detect if a collision has occurred, however this was not implemented as providing both collision detection components separately allows a snake to learn separate behaviors depending on collision type, for example chasing it's tail or circling the grid perimeter.

The second method of collision sensing is referred to as `space_aware` and was used by both `space_aware-standard` and `space_aware-n+n`. This method was created to attempt to prevent the snake being trapped with no space to go. To create this sensing function, a recursive search was performed for a given position that counted all available spaces from that position. This search is performed at each of the four possible positions that the snake can move to (up, down, left and right).

1.5 Population Selection Between Generations

Two selection methods have been developed for this report. Both generation to generation selection methods use a tournament selection method with a size of 3 to select individuals to evolve. Where the two selection methods standard and n+n differ is their assignment of the population to be used for the next generation. Standard evolution simply uses tournament selection to select a full population of individuals, these individuals are then evaluated and all selected individuals are assigned to the population.

n+n evolution, as discussed by Yeh et al [2], uses the concept of preserving best individuals from earlier generations without mutation. This is performed by tournament selecting a full population of individuals that are then evaluated. n+n evolution combines the best half of the previous generation population with the best half of these newly evaluated individuals, to preserve individuals with promising weighting from being overwhelmed by random mutations.

1.6 Fitness, Mutation and Crossover

Multiple fitness functions were tested before settling on one fitness function for all snakes that gave best performance. This fitness function was detailed by Haber [3] and is of the form $(\texttt{score}^3) * \texttt{timeAlive}$, where `timeAlive` is a counter incremented each move. A starvation function of $\min(5 * \texttt{snakeLength}, 200)$ was implemented alongside this to prevent snakes developing behavior that maximises time alive over score. An additional variable `timeSinceEaten` was incremented each move and reset to 0 when food was eaten by the snake, this was the variable compared to in the starvation function. If a snake died to starvation, `timeSinceEaten` was subtracted from `timeAlive` to discourage future starvation promoting behaviour.

Mutation probability was explored in two differing ways, the first being `MUTPB` which is the probability that an individual is selected for mutation. The second was `indpb`, which is the probability that a single attribute of an individual is mutated. Gaussian mutation was chosen to mutate with, and through preliminary testing ideal values were found to be $\texttt{MUTPB} = 1, \texttt{indpb} = 0.1$.

Through preliminary testing, both single and two-point crossover were found to harm the evolution

of this implementation of snake AI evolution, and were subsequently removed from final versions of all algorithms by setting the constant `CXPB` to 0, for a 0 probability of an individual experiencing crossover.

2 Results

2.1 Focus of Testing

Testing will focus on performance of algorithms containing different methods of evolution selection and collision sensing. Multiple different features have been investigated for this report; standard evolution against n+n evolution, and local sensing against space aware sensing. These two factors combine to provide four statistical comparisons between algorithms.

All other components of algorithm variants are kept the same, e.g. fitness functions, mutation probability.

2.2 Testing Methods and Data Collection

Each algorithm was ran 10 times with a population of 200 for 500 generations, an average of these runs was then computed to provide a fair graphical representation of the performance of each algorithm.

Statistical analysis involved performing a Mann-Whitney-U test at each generation, where a sample consisted of the average value for each run of the algorithm to give a total of 10 samples per generation, per algorithm. A difference of means m was also calculated at each generation. This p, U, m can then be averaged for all generations to give an average statistic over all generations.

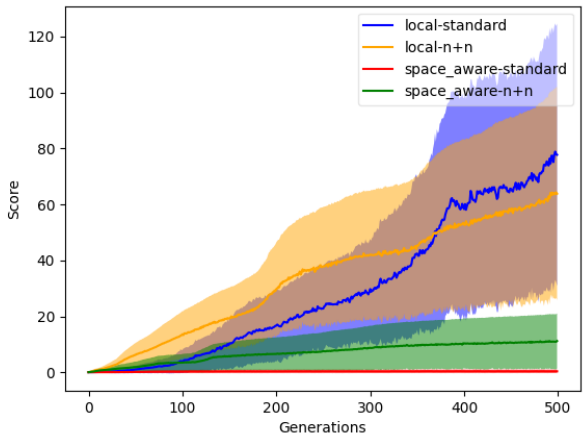

Figure 1: All algorithm variants compared over 500 gens

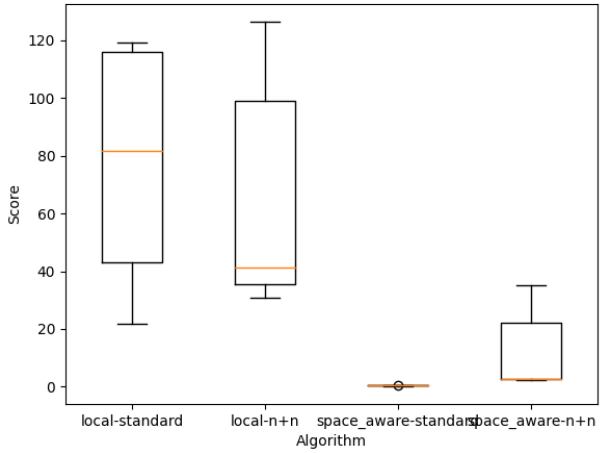

Figure 2: Boxplot of final generation for each algorithm variant

2.3 Effects of Changing Evolutionary Selection Method

Figure 3 compares standard evolution to n+n for local collision sensing, while `local-standard` was the prevailing algorithm after 500 generations, `local-n+n` is consistently performs better from generation 0 until a crossover point at approximately 50 score. Average Mann-Whitney-U and difference of means are:

$$U = 32.8, p = 0.39137, m = 8.9$$

As $p > 0.05$, changing evolution selection methods for local sensing is not statistically significant.

Figure 4 compares standard to n+n evolution for space aware collision sensing. As shown in the graph, n+n performed significantly better than standard evolution, though standard deviation was large. Standard evolution struggled to make any progress throughout generations, with a consistently average score throughout generations of 0. Average Mann-Whitney-U and difference of means are:

$$U = 0.0, p = 0.00018, m = 6.7$$

Changing evolution selection methods for space aware sensing is statistically significant as $p < 0.05$. While the difference of means m does not immediately suggest a large effect size, when considering figure 4 the data shows that n+n evolution allows space aware sensing to be a viable collision detection method unlike standard evolution. This is likely due to n+n evolution being significantly more sensitive to marginal improvements in fitness within a generation, as n+n is designed to preserve best individuals, while standard evolution leaves the survival of an individual down to a weighted probability.

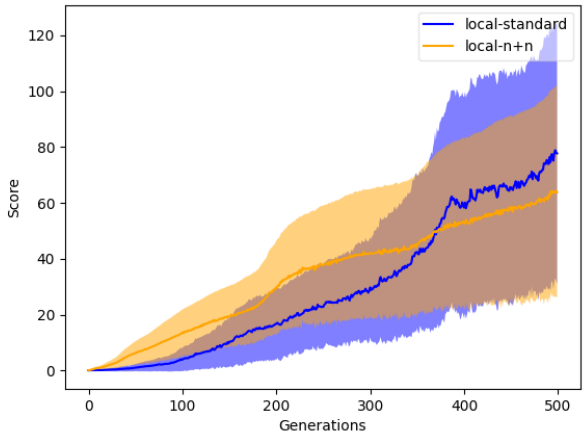

Figure 3: `local-standard` compared to `local-n+n` over 500 gens

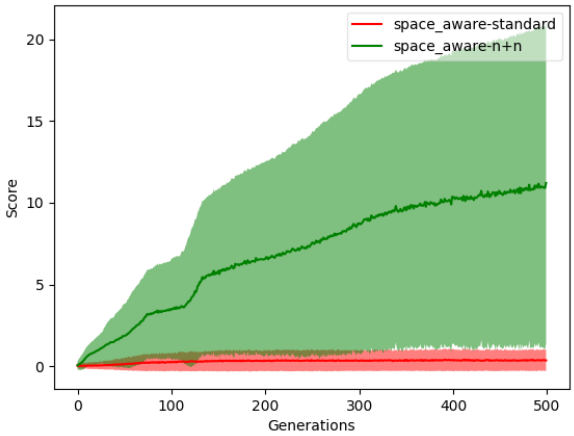

Figure 4: `space_aware-standard` compared to `space_aware-n+n` over 500 gens

2.4 Effects of Changing Collision Sensing Method

Figure 5 compares local collision sensing to space aware collision sensing using standard evolution. Local collision sensing managed to successfully learn how to achieve high scores, with best individuals even able to complete the game by filling every square. Space aware collision sensing was unsuccessful in any of the 10 runs to evolve any sustainable method of finding food and increasing score. Average Mann-Whitney-U and difference of means are:

$$U = 0.2, p = 0.00063, m = 28.7$$

As $p < 0.05$, these results are statistically significant. This is due to `space_aware-standard` not successfully evolving any method of increasing score, while `local-standard` was extremely successful. An average difference of means of 28.7 combined with a U value of 0.2 shows a large effect size.

Figure 6 compares also compares local collision sensing to space aware collision sensing, however n+n evolution is used to evolve the population. Average Mann-Whitney-U and difference of means are:

$$U = 9.0, p = 0.00404, m = 26.5$$

These results are statistically significant $(p < 0.05)$. In contrast to figure 5, n+n evolution has successfully trained space aware collision to be able to achieve a non-zero average score. Regardless of this, space aware collision sensing still performed significantly worse than local sensing, with an average

difference of means of 26.5. The effect size of this statistical comparison is large, as evidenced by a U value of 9.0. These difference of means and U value are influenced by early generations, however local sensing quickly outperforms space aware sensing as evidenced in figure 6.

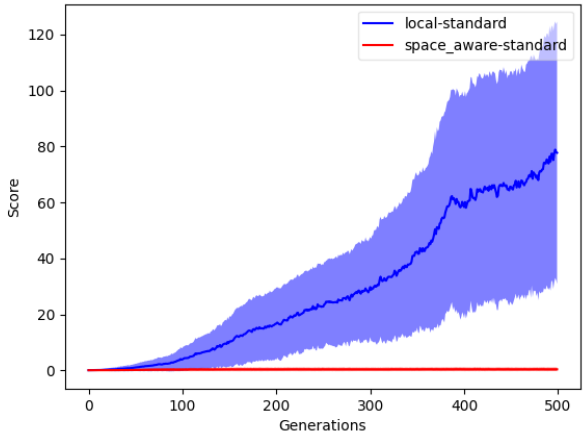

Figure 5: `local-standard` compared to `space_aware-standard` over 500 gens

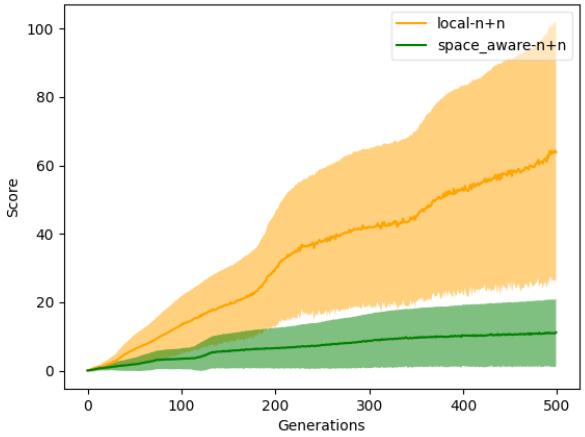

Figure 6: `local-n+n` compared to `space_aware-n+n` over 500 gens

3 Conclusions

Results clearly show that local sensing is the superior collision sensing method to space aware (search for available space) sensing. This is likely due to space aware sensing being significantly more sensitive and therefore easy for progress to be overwhelmed by random mutations, as evidenced by `space_aware-n+n` being able to successfully learn how to gain an average of approximately 20 score, while `space_aware-standard` struggled to make any progress at all. An added benefit of local sensing performing significantly better compared to space aware relates to time taken to run a generation. For an example 50 generations evolution of a population of 200 individuals, local sensing completed all generation evolutions in 6.2 seconds, while space aware took 51.8 seconds. This 8.4x slowdown in evolution drastically increases training time in later generations.

With our best sensing method of local collision sensing selected, conclusions can be drawn about our chosen evolutionary selection methods. Unfortunately the two datasets (`local-standard` and `local-n+n`) were found to be not statistically significant. While the datasets were not statistically significant, future work should focus on longer generation testing, with the goal of finding if these two datasets diverge in later generations. At the stopping point of 500 generations in this report, both datasets showed no signs of reaching a plateau. Logic suggests that n+n evolution should plateau at a higher score than standard evolution due to its reduced susceptibility to mutations that decrease score, however without data to show this, no conclusion regarding this can be drawn from this report.

Additionally future work may wish to investigate the effects of implementing crossover, or changing mutation probability values on each of the algorithms detailed in this report. All algorithms in this report used the same standard values as control variables, however each algorithm may benefit from tuning these parameters.

Provided final solution code is of the `local-standard` algorithm, with a best individual provided pickled in `bestSnake.ind` that can be ran by file `runSavedBestSnake.py`.

References

[1] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[2] J.-F. Yeh, P.-H. Su, S.-H. Huang, and T.-C. Chiang, *Snake game ai: Movement rating functions and evolutionary algorithm-based optimization*, Nov. 2016. DOI: 10.13140/RG.2.2.33593.36969.

[3] C. Haber, *The ai for snake game chronicles*. [Online]. Available: https://craighaber.github.io/AI-for-Snake-Game/website_files/index.html.

Appendix

A Provided Files and Changes to Provided Code

All supplied code files are detailed as follows:

A.1 code/displayGame.py

This file contains the `DisplayGame` class, the only change to this class is changing the snake body colour to be only green to remove any confusion on which way the snake is facing.

A.2 code/neuralNetwork.py

This file contains the `NeuralNetwork` class, this is the neural network that is evolved by the snake.

A.3 code/runGame.py

This file contains the `runGame` function, other than the code added to facilitate AI snake control into this, other changes have been made. A guard was added to the check of if the snake has eaten food to see if the snake has completed the game. Game completion is defined as there being no available spaces on the board to place food as the snake is taking up all available grid positions. There are also changes to the clean-up of the Python turtle to prevent Python kernel crash.

A.4 code/snake.py

This file contains the `snake` class. While sensing functions have been added to this class, the only change to initially provided functions has been to use `self.XSIZE` and `self.YSIZE` instead of the global `XSIZE`, `YSIZE`.

A.5 code/runSavedBestSnake.py

This file will load a pickled individual and run the game graphically. Running this file will automatically load `bestSnake.ind` and show the user the best snake playing the snake game.

A.6 code/trainSnake.py

This file will train a population of 200 individuals for 500 generations using the `local-standard` algorithm developed in this report. Population data will be saved as `snake.pop`, best individual as `snake.ind`, and a dictionary with keys `fitnessLogbook` and `scoreLogbook` that correspond to respective logbooks are saved as `snake.stats`.

A.7 code/bestSnake.ind

This is a pickled form of the best trained individual in population `code/bestSnake.pop`, trained for 500 generations. Can be seen playing the game by running `code/runSavedBestSnake.py`.

A.8 code/bestSnake.pop

This is a pickled form of a population trained for 500 generations that contains `code/bestSnake.ind`.

A.9 code/bestSnake.stats

This is a pickled form of a dictionary that contains 500 generations of fitness and score logbook data to be used for graphing. Data is of `code/bestSnake.pop`, and is of form of a dictionary with keys `fitnessLogbook` and `scoreLogbook` that correspond to respective logbooks. These logbooks contain `min`, `max`, `avg` and `std` values.

END OF PAPER