Developing An Agent To Play Snake Using Evolutionary Computation

Author:
Y3507677

Module:
Evolutionary Computation

January 24, 2019

Abstract

This paper covers the development and implementation of an agent that is tasked with maximising it's score while playing the traditional snake game. The agent itself only has sensing functions, four movement functions and must develop it's own strategy to solve the task at hand. The algorithm is developed using Python and the Distributed Evolutionary Algorithms in Python (DEAP) evolutionary computation framework.

Contents

I. Introduction

First introduced as a multiplayer game, the origins of Snake date back to another game known as 'Blockade' that was introduced in 1976 by Gremlin Industries [1]. Following the release of Blockade, several notable games built upon the foundation of it: the first PC version (Tandy TRS-80) was Peter Trefonas's 'Worm' in 1978 [2], and Nokia's 'Snake'.

Nokia's version of the original Blockade game was developed by Taneli Armanto and released in 1977 on the Nokia 6110 [3]. Following this, Snake has become a widely known, simple computer game where the player is tasked with achieving the highest score possible using the snake. The concept is simple, the snake is always moving, tasked with eating the fruit that is available on the grid and to not hit in to itself or the walls that surround the map. For every piece of food that the snake eats it's tail grows longer and thus the difficulty increases proportionally to the score achieved. In Snake, there is no concept of lives and if the snake hits a wall or in to itself then the game ends.

The implemented version of snake has similar conditions to the version developed by Armanto, however, a 14x14 grid is used and the snake agent starts with a length of 3. Therefore, the maximum possible score is 193. Like the original versions of the game, the snake is always moving, can move in four absolute directions and food cannot be placed at a coordinate that the snake's body is occupying.

To ensure that no agent is assigned a fitness score as a result of luck, Ehlis averaged an agent's score over multiple runs of the game and assigned the fitness value as this result. In addition to ensuring that the score is fair, an average of multiple runs also ensures that the agent is performing effectively. This helps to reduce elitism in the population from a lucky run.

II. Existing work

A. Genetic Programming

Some existing work has been undertaken that applied genetic programming to the snake game, by Tobin Ehlis [4] in 2000. Ehlis focused on trying to maximise the score of the agents through the use of a fitness function that measured it's score based on food eaten; a round being one attempt at eating the food. Ehlis's work performed well and achieved a maximum score of 211 out of 220; 96%.

Ehlis's implementation developed a number of strategies to assertain the highest score possible. Some developed strategies which would hug the wall and zig zag along it to ensure that it could move forward uninterupted. However, before moving towards a fitness function that did indeed measure how much of the grid the agent had covered, score was taken in to consideration. This resulted in solutions that would plateau in later generations due to the snake colliding with it's body.

Ehlis observed that as generations progressed, they took no interest in the functions that observed if there was food nearby. The removal of these functions, allowed agents to developed efficient solutions that resulted in high scores. The agents utilised the functions that sensed if there was danger two steps away from it's current position. The agents followed a circular pattern to navigate around the grid. To run Ehlis's algorithm took around 20 hours on average. While this will not be possible given time limitations, it is hopeful that the solutions here will run much faster given the advances in computational power since 2000.

B. Machine Learning

Other artificial intelligence fields have produced work that produced agents that can play the snake game also, Wang et al. [5] developed autonomous agents using deep reinforcement learning that employed a fitness function which measured the score and survival time; both as maximisation functions. Their model and fitness functions produced good results and they introduced an interesting notion of warming up the population before the training commenced. During this time, the agents randomly activated functions and drew some relationships before the training was started. This provided the starting population with the ability to navigate the grid before developing a strategy.

III. Design and Implementation

A. Methodology

To solve the aforementioned problem, all code was written using Python 3.6.7 and DEAP version 1.2.2 in PyCharm 2018.3.3 Professional Edition (build: 183.5153.39). Code was executed on Ubuntu 18.04, on a Ryzen 5 1600 @ 3.85 GHz, 8GB DDR4 and a 500GB SSD, as well as being tested on the Computer Science department's computers to ensure that it worked effectively on university computers.

Due to the size of the search space, it was made a requirement that DEAP's multiprocessing module was used to help distribute the workload across all available threads. An early incarnation of the solution enabled the multiprocessing pool and had a main loop similar to what was available in DEAP's One Max Problem tutorial [6]. However, throughout the development and testing process, system resources were always monitored to ensure that the maximum available resources were being provided to the process and it was noticed that this solution only used one thread at any given time. Upon further investigation, it was noticed that DEAP's `eaSimple` function used the multiprocessing module correctly. In order to maximise development time, the `eaSimple` code was extracted and used as the base for running the GP algorithm. The root cause of this problem was not isolated but this was deemed as a solution to the issue. Use of the multiprocessing module lowered the average run time of the algorithm from hours to minutes.

The use of graphics processing units (GPUs) to minimise the time required to find an optimal solution and accelerate development time was investigated but no available solution was found. Machine learning frameworks such as TensorFlow [7], PyTorch [8], and Caffe [9] utilise GPUs but DEAP does not. While there have been a number of requests for this [10], DEAP does not currently support such technologies and the multiprocessing module is the only technology available to expedite convergence times.

B. Genetic Algorithm

C. Approach

It was decided that the A.I to be implemented should focus on maximising the score while using the least amount of possible computational resources; in a bid to reduce the overall time that it would take the GP to converge on an acceptable solution. To achieve this goal, care was taken to ensure that functions are as efficient as possible, the function set made available to the G.A was as small as possible and height limits placed on trees.

Previous approaches, such as Ehlis' [4] focused on maximising the score through the coverage of the snake. The approached implemented here focuses on allowing the snake to generate it's own strategy and seeing if any new novel behaviour can be found. GP is, however, still implemented as the. GP was used as the implementation strategy as it most closely suits the problem that is to be solved. Similarly to Ehlis's implementation, multiple runs of the game are averaged to see if the agent's score was a result of the fruit being placed nearby to the snake's head.

D. Fitness Function

As the search space is so large, it is important to consider how to implement the fitness function for this problem with great care. For the problem at hand, a constraint exists that the snake does not die and an objective exists that the snake should aim to get the highest score possible. The constraint, however, decays as the score increases and it is important that the fitness function reflects this; if the snake dies after the first round then it has done poorly but if it achieves a score of 80 then it has done well.

While designing the fitness function, multiple approaches were considered and their design greatly influenced the strategy that the snake would adopt to solve the problem.

The approaches considered were as follows:

- **Score-driven**. A purely score driven function would allow for the snake to develop it's own strategy but could result in a far larger training time and a strategy that starts to struggle as time-complexity increases.
- **Non-greedy**. The fitness function should evaluate both the score that was achieved as well as the number of steps it took overall. Both the score and steps should be maximised as this will encourage the population to take the longest path possible in between eating the fruit. Such an approach could lead to strategies similar to the Hamilton solver or new novel behaviour. Agents that do take this path will be under a constraint that terminates the game if the number of steps exceeds the total number of points on the grid; this will stop agents from spinning in circles.
- **Non-greedy and unaware of food**. To further increase the promotion of developing a strategy that is focused on taking the longest route the above approach could be strengthened by removing any food sensing functions that are available. This would encourage the snake to learn the longest possible route around the grid and as a result of taking this route it would naturally encounter the food.
- **Greedy**. A non-greedy solver would rely on maximising the score and minimising the number of steps in between eating the fruit. It is hypothesised that this would lead to a sub-optimal strategy once the size of the snake increases.
- **Novelty search**. Lehman [11] proposed the notion of rewarding an individual on the basis of how novel it's solution was when weighed against the population. The implementation of this was considered as a possible fitness metric, however, not enough metrics of novel behaviour could be found. Some considered were: how novel the route the path that the agent had taken each round was, taking the shortest or longest route possible. Though these were valid metrics, the resources required to compute them each round were deemed too expensive.

As a result of the random placing of the fruit, it was decided that multiple runs would take place to ensure that the snake does not get a high score as a result of where the fruit spawns. Averaging both the score and the number of steps will help to normalise the fitness values.

Upon considering the approaches that could be adopted, it was decided to implement the non-greedy approach to see if any novel behaviour could be found as opposed to encouraging the snake to develop behaviour similar to a Hamilton cycle.

Therefore, the fitness function is:

$$Fitness = \begin{cases} \left\lceil \frac{\sum_{1}^{n_{\text{evals}}} score^2}{n_{\text{evals}}} \right\rceil & if\ score > 0\ otherwise - 10 \\ \\ \left\lceil \frac{\sum_{1}^{n_{\text{evals}}} steps}{n_{\text{evals}}} \right\rceil & if\ steps > 0\ otherwise - 10 \end{cases} \tag{1}$$

Note: the mathematical ceiling notation has been used here to denote that the variable is to be maximised

E. Primitive Set

One of the most significant differences that could be made to the overall performance of the snake was the function set available to the agent. While developing the function set that would be available, it was important to realise how the functions would be used and the possible (undesired) outcomes that could arise. For example, what would happen if the snake was to turn right when it was travelling left? It would run itself over. This notion resulted in the addition of four new primitive functions that would be made available; `if_moving_DIRECTION`. An alternative option was considered that prevented the agent from attempting to run itself over but it was instead decided to run the GP for longer and allow it to discover this correlation itself.

Primitive	**Terminal**
`if_food_up`	`change_dir_up`
`if_food_right`	`change_dir_right`
`if_food_down`	`change_dir_down`
`if_food_left`	`change_dir_left`
`if_danger_up`	
`if_danger_right`	
`if_danger_down`	
`if_danger_left`	
`if_moving_down`	
`if_moving_up`	
`if_moving_right`	
`if_moving_left`	

TABLE I
PRIMITIVE SET FUNCTIONS IMPLEMENTED

The act of deciding whether or not to add functionality in to existing functions or create new ones became an important trade-off between guiding the snake or allowing it to develop novel behaviour itself. In addition to this trade-off, there is an increase in the size of the tree that is developed when more functions are added.

Food sensing functions sense food along the entire axis that the snake is positioned on and looking at the direction of the food. This allows the snake to be guided towards the food. An alternative would be to only have a range of one coordinate in the direction that the snake is looking and this may be considered future work.

In opposition to the food sensing functions, the danger sensing functions only have a range of one. If the functions had a range of the entire axis then the agent would always sense danger ahead

1) Axis Sensing vs Neighbour Sensing:

While developing the sensing functions, relative and absolute directions were both considered and it was decided that absolute directions would be a more appropriate system to use due to the simplicity of it. The use of relative directions results in lots of checks in the code to check the current heading. While the use of relative directions does indeed result in a smaller function set, absolute directions result in simpler code that is more maintainable and readable.

To perform a test of axis vs neighbour sensing, a population of 500 was selected and 300 generations were run with a fitness function that is maximising both the average score and average number of steps over 3 runs of the game. Using double selection tournament with a fitness size of 3, parsimony size of 1.1 and evaluating the fitness first. One point crossover and uniform mutation was selected with a height limit of 12 on both. To ensure the results were as accurate as possible, five runs were done of each and random seeds provided each time. Table II details the results of this comparison.

The results clearly show that absolute directions are a more optimal approach to adopt in the GA, even though the function set is larger. While both the average score and standard deviation are higher when using absolute directions the maximum observed fitness score is significantly further away from the average when this approach is adopted. This demonstrates the need to run the GP for more generations when a large function set is used.

	Neighbour Sensing	Axis Sensing
Avg Fitness	438.209	1678.73
Avg Score $\lceil\sqrt{x}\rceil$	21	41
Standard Deviation	265.465	781.779
Maximum Fitness	916.667	3147
Maximum Score $\lceil\sqrt{x}\rceil$	30	56

TABLE II
AXIS SENSING VS NEIGHBOUR SENSING COMPARISON

Stating a null hypothesis (H_0) that that the results would be similar in both axis sensing of neighbouring a Mann-Whitney U test was run with results of: $U = 45916$ and $p = 1.445e - 31$. As the $p < 0.05$ we can reject the null hypothesis and use the axis-sensing function set since it clearly leads to a higher fitness value.

F. Population

1) Initialisation:

To initialise the population, individuals were set to grow their tree size in the range of 1 to 3. This allowed encouraged the population to learn the basic functions required to survive in the world before developing complex solutions, as well as decreasing the time required to converge on a solution.

In order to determine the optimal population size, multiple tests were performed over 300 generations of population sizes of: 100, 300, 500, 1000, 2000. The results, as detailed in appendix: A, show that the population size of 1000 has a clear upwards trend in the maximum fitness score over the 300 generations that it was run against. Unlike the lower population sizes where the fitness score plateaus after generation 150. However, the lower population sizes have a higher fitness score in the earlier generations when compared to the population sizes of 1000 and 2000. As the time complexity with a population size that is greater than 500 is so high it was decided upon to initialise the population with a size of 300. This population size performs better than 500 and still achieves a high fitness value in the later generations. One could deem 300 to be the optimal population size when comparing it to the computational time and resources available. Gotshall and Rylander [12] concluded that ”the greater the population size the greater the chance that the initial state of the population will contain a chromosome representing the optimal solution” and so the largest population size possible will be used given time and computational resource constraints.

The population size will remain consistent across all generations, though some studies have found that results can improve and a reduction in computational resources can be found when using a dynamic population size in certain situations.

2) Bloat Control:
Due to the nature of genetic programming, an individuals size can grow exponentially until Python's interpreter parser stack limit is reached (92-99)[13] unless control measures are put in place. DEAP offers one of the most common approaches to bloat control [14, p411], maximum depth restrictions through the use of decorators on the toolbox. A height and size limit of 14 was ascertained through trial and error, and offered a acceptable convergence times and lower system resource utilisation. As the fruit is randomly placed each time, there is no requirement to be concerned about over-fitting.

In addition to decorators, a `selDoubleTournament` was used to aid in reducing the bloat of an agent's solution. This introduces another metric that can be used to gauge an agent's solution to the problem at hand; tree size. On average, experiments revealed that a parsimony selection ($\mathrm{S_p}$) size of near to 1 (close to random) returned the highest fitness values. $\mathrm{S_p}$ closer to 2 returned no additional benefit of using `selDoubleTournament` and a solution could be ascertained quicker using `selTournament` without the additional overhead. Thus, $\mathrm{S_p} = 1.05$ was used.

3) Crossover:
DEAP offers two mutation strategies: one point, and one point leaf biased. One point leaf biased produced increased the average fitness score at a lower generation and a terminal selection probability (`termpb`) value of 0.1 was used. Due to the low proportion of terminal nodes to primitive nodes, a high `termpb` value caused the agent to change it's direction more frequently and produced undesired movements.

4) Mutation: To keep inline with wanting the agent to develop novel strategies in it's search for a solution, uniform mutation was used as the mutation strategy. Uniform mutation also resulted in the fastest convergence time to an acceptable solution. While node replacement does mutate the node with another individual's solution, an individual's solution path to the terminal node is so tightly coupled that selecting another individual's solution resulted in ineffective solutions after mutation. Thus, uniform mutation was deemed the most effective.

IV. Results

Overall, the implementation produced good results; see table III. Scores had a wide distribution, test runs after training produced acceptable results and interesting strategies. Some runs produced strategies which would zigzag along the sides of the walls to flatten itself before then going off to fetch the food. The leading cause of failure appeared to be the snake attempting to go in to a space where it would not be able to get back out of. While these strategies are valid, they do not lead to high enough scores and thus some optimisation is required to the algorithm.

Appendix F shows the agent's function tree that was developed to solve the problem. With a maximum height of 8, for each move the snake makes there are a maximum of 8 functions that may be executed. DEAP's offering of decorators to limit height and size limit has allowed the algorithm to develop a robust solution that utilises the function set wisely in order to maximise it's fitness score. As Luke et al. [14] concluded, bloat control may not increase fitness but it controls the size of the solution. This has been clear here, as the snake could have added every one of it's movements to navigate around the grid to the tree. This particular kind of problem has strongly benefited from bloat control.

A. Optimisation

The results in appendix D show that after 350 generations the current implementation starts to struggle in achieving a higher score. This is believed to be as a result of the danger sensing

Metric	Value
Std. deviation of fitness	672.98
Average fitness	697.24
Maximum fitness	2601
Maximum score	51

TABLE III
RESULTS

functions not providing information to the snake that the available path to the food will trap it; the snake is sensing food, seeing that there is no danger ahead and the proceeding to take the available path to it and then becoming trapped. With the existing function set, there is no way for the snake to tell that it will become trapped. Possibly introducing an if statement function that has an arity of three would allow the snake to develop a more complex solution, however, this would still be less than optimal.

1) Flood-fill:

One optimisation strategy was to increase the sensing abilities of the danger sensing functions through the introduction of an `if_trapped` function. This function checks to see if the next position of the snake's head is in a position where it would become trapped, given it's heading. This works in the following way:

- Parse the grid. Converting 'walls' (snake's body and boundaries) to 0s and then fill inside those walls to 1s. Small numbers were chosen as the character of choice to reduce the overall time it takes to execute this algorithm.
- Subtract 1 from the total number of rooms to exclude a global fill of the grid. If any rooms exist, then continue on to the next part.
- Given the snake's current heading, find the nearest room to the snake within a distance of 1. If a room exists, then find the size of that room and negate the result of whether or not the snake can fit in to that room.

The results, as detailed in table IV show a clear increase in the performance of the algorithm with the flood-filling implemented. However, the standard deviation and average fitness are quite close in value and this raises the point that further generations should be run to ascertain the true performance of the feature. This could be because there is a primitive node that is always not providing much input to the solution in the earlier generations.

The flood-filling function was trailed in the danger sensing functions, however, there is already a large enough performance hit when using this algorithm and implementing it in four functions slowed the algorithm down significantly. This should be deemed future work to discover how this algorithm affects the performance of the agent.

	Pre-optimisation	Optimised
Std. deviation of fitness	672.98	830.18
Average fitness	697.24	768.95
Maximum Fitness	2601	4624
Maximum Score	51	68

TABLE IV
COMPARISON OF PRE-OPTIMISATION AND POST-OPTIMISATION

The graphed results of this optimisation are available alongside a user friendly version of the above algorithm is available in appendix B. Following implementing this optimisation, a noticeable increase in the performance of the agent was observed. The results, as detailed in D show a noticeable increase in the performance of the agents over the generations. A maximum score of 68 was achieved when using the flood-fill algorithm in contrast to 51 without it; a 16% increase score. The distribution of scores as detailed in appendix E shows that a peak at the average score. This details a more consistent ability of the agent, however, further runs of the algorithm need to be performed with a higher number of generations to see if this distribution shifts to a higher score. This is because the flood-filling algorithm has little affect on snakes of a short length and if agents have weened out the function due to it providing no value in early generations, then a mutation is required that contains it before it has a large enough impact on the agent.

V. Conclusion

A. Findings

This report has detailed the implementation, findings and optimisations to implement an agent that can play the Snake game effectively. Overall, the initial goal of developing an effective agent was met and acceptable scores were achieved. While these results are acceptable, more consideration and testing should be performed in the future before settling on the values to be used for the algorithms. For example, once the flood fill was implemented, the number of generations should have been increased to allow agents to start using the new function available. While it is clear that it has improved the average fitness score, it would seem that not enough generations have been run in order to ascertain how effective the function is at higher scores. Further mutations are required in order to utilise the function correctly.

B. Future Work

While the approach implemented has achieved an acceptable score, it did not reach half of the maximum score that it could have done. This leaves room for improvement and some alternative methods are proposed below.

- **Removing vision from the snake**. It is clear that the snake starts to struggle increasing it's score at scores past 70 and so an alternative fitness function could be implemented that encourages the snake to take the longest route possible. Such an implementation would disregard any food sensing abilities and focus only on avoiding danger and following the longest path possible. In the course of the snake covering every point on the grid, it would naturally come across food and achieve the highest score possible. In addition to rewarding the snake for taking the longest possible path, an additional fitness value could be introduced that encourages the snake to take this path from the coordinate (0, 0) such that it starts it cycle from the corner of the grid; aiding the snake in it's search for the perfect solution. While this could have been implemented from the start, it has been interesting seeing the snake develop it's own strategies for achieving it's high score.
- **Co-evolution: snake and food**. Another implementation approach could have been to evolve the snake and food as two separate agents. The snake, which would follow a similar implementation strategy, and the food which would evolve to spawn in the least accessible position on the grid.
- **Expanding on Want et al's work**. The population could be warmed up with a limited function set until every agent, for example, had achieved a score of one. This population could then be used as the starting population in a bid to attempt to start training with a population that already knew how to navigate the map.

References

[1] "Blockade [model 807-0001]." [Online]. Available: https://www.arcade-history.com/?n=blockade&page=detail&id=287

[2] M. Longshaw, "Retro corner: 'snake'," Apr 2011. [Online]. Available: https://www.digitalspy.com/gaming/retro-gaming/a313438/retro-corner-snake/

[3] "History of nokia part 2: Snake," Jan 2009. [Online]. Available: https://web.archive.org/web/20110723064106/http://conversations.nokia.com/2009/01/20/history-of-nokia-part-2-snake/

[4] T. Ehlis, "Application of genetic programming to the snake game," Aug 2000. [Online]. Available: https://www.gamedev.net/articles/programming/artificial-intelligence/application-of-genetic-programming-to-the-snake-r1175/

[5] Z. Wei, D. Wang, M. Zhang, A.-H. Tan, C. Miao, and Y. Zhou, "Autonomous agents in snake game via deep reinforcement learning," *2018 IEEE International Conference on Agents (ICA)*, 2018. [Online]. Available: https://www.researchgate.net/publication/327638529_Autonomous_Agents_in_Snake_Game_via_Deep_Reinforcement_Learning

[6] "One max problem." [Online]. Available: https://deap.readthedocs.io/en/master/examples/ga_onemax.html

[7] "Tensorflow." [Online]. Available: https://www.tensorflow.org/

[8] "Pytorch." [Online]. Available: https://pytorch.org/

[9] "Caffe." [Online]. Available: http://caffe.berkeleyvision.org/

[10] [Online]. Available: https://groups.google.com/forum/#!topic/deap-users/iz41IHgCDvc

[11] J. Lehman, "Evolution through the search for novely," vol. 9, 2012. [Online]. Available: http://joellehman.com/lehman-dissertation.pdf

[12] H. Yong, "Optimal population size for partheno-genetic algorithm," *2007 Chinese Control Conference*, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.2431&rep=rep1&type=pdf

[13] "Genetic programming." [Online]. Available: https://deap.readthedocs.io/en/master/tutorials/advanced/gp.html

[14] S. Luke and L. Panait, "Fighting bloat with nonparametric parsimony pressure," *Parallel Problem Solving from Nature PPSN VII Lecture Notes in Computer Science*, p. 411 421, 2002. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45712-7_40

APPENDIX A
POPULATION SIZE

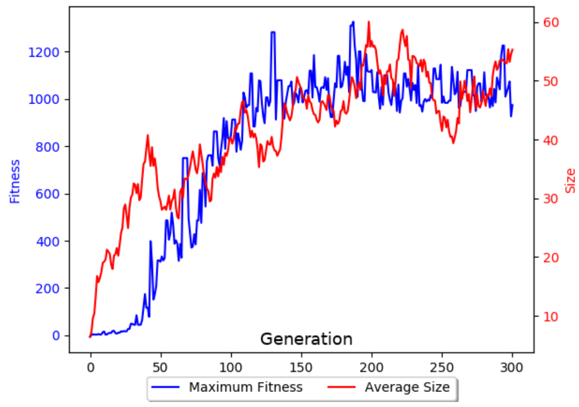


Fig. 1. Population size of 100 over 300 generations

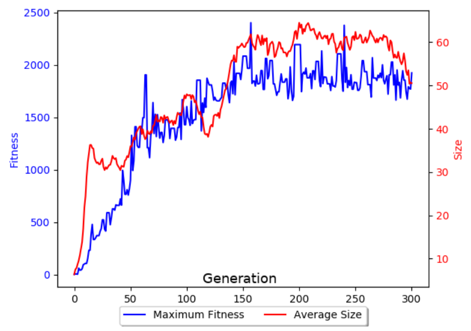


Fig. 2. Population size of 300 over 300 generations

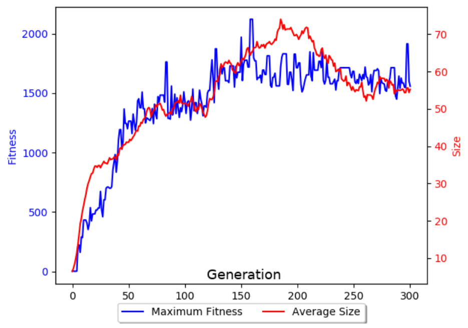


Fig. 3. Population size of 500 over 300 generations

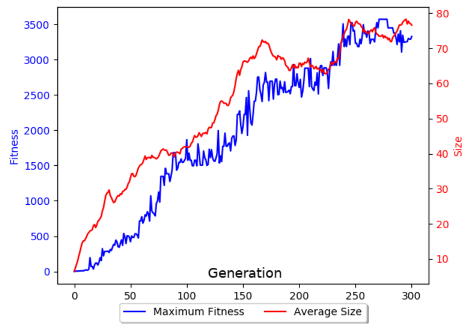


Fig. 4. Population size of 1000 over 300 generations

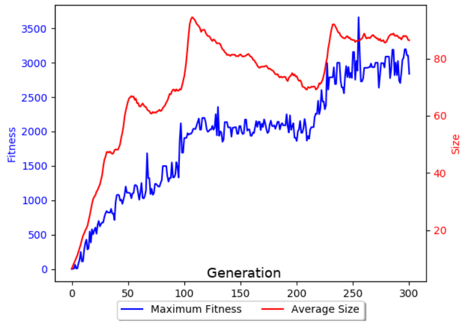
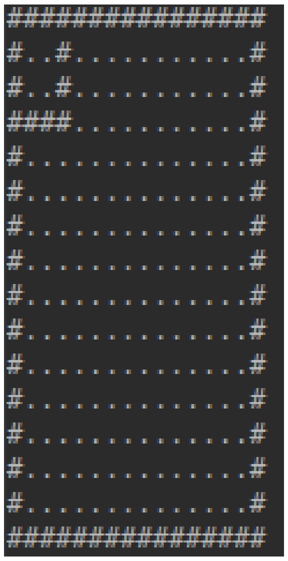


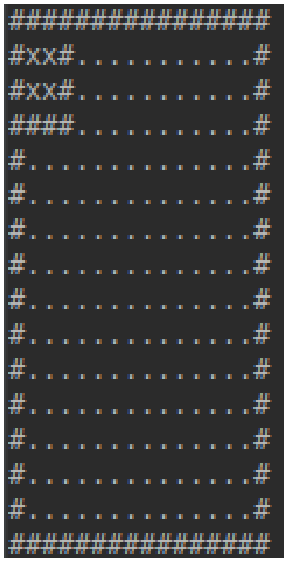
Fig. 5. Population size of 2000 over 300 generations

Appendix B
Flood-fill



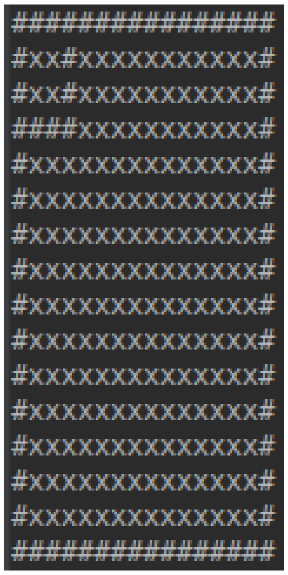
```
################
#..#...........#
#..#...........#
####...........#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
################
```

Fig. 6. Layout that will be flood-filled. The snake is at the top left.



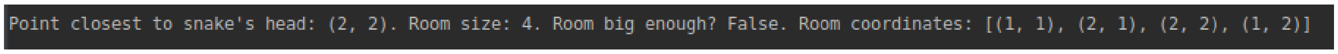
```
################
#xx#...........#
#xx#...........#
####...........#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
#..............#
################
```

Fig. 7. After the first flood-fill has been performed.



```
################
#xx#xxxxxxxxxxx#
#xx#xxxxxxxxxxx#
####xxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
#xxxxxxxxxxxxxx#
################
```

Fig. 8. After the second flood-fill has been performed.



```
Point closest to snake's head: (2, 2). Room size: 4. Room big enough? False. Room coordinates: [(1, 1), (2, 1), (2, 2), (1, 2)]
```

Fig. 9. A user-friendly output from the algorithm displaying that the room to the left of the snake's heading is not large enough for the snake. The outputted coordinates are off-by-one as the boundaries are automatically inserted.

APPENDIX C
ALGORITHM VALUES

Type	**Value**
`Number of generations`	500
`Crossover probability`	0.8
`Mutation probability`	0.2
`Population size`	500
`Initialisation strategy`	`grow` min: 1, max: 3
`Selection`	`double tournament` fitness size: 3, parsimony size: 1.1, fitness first
`Mating strategy`	one point point leaf biased, terminal selection probability: 0.2
`Expression generation strategy`	`full`, min: 0, max: 1
`Mutation strategy`	`uniform`
`Mating height limit`	12
`Mutation height limit`	12
`Number of games run per evaluation`	3
`Fitness function`	$(\mathrm{score}^2/\mathrm{noGames}), (\mathrm{steps}/\mathrm{noGames})$ - both maximised

TABLE V
ALGORITHM VALUES

Appendix D
Results

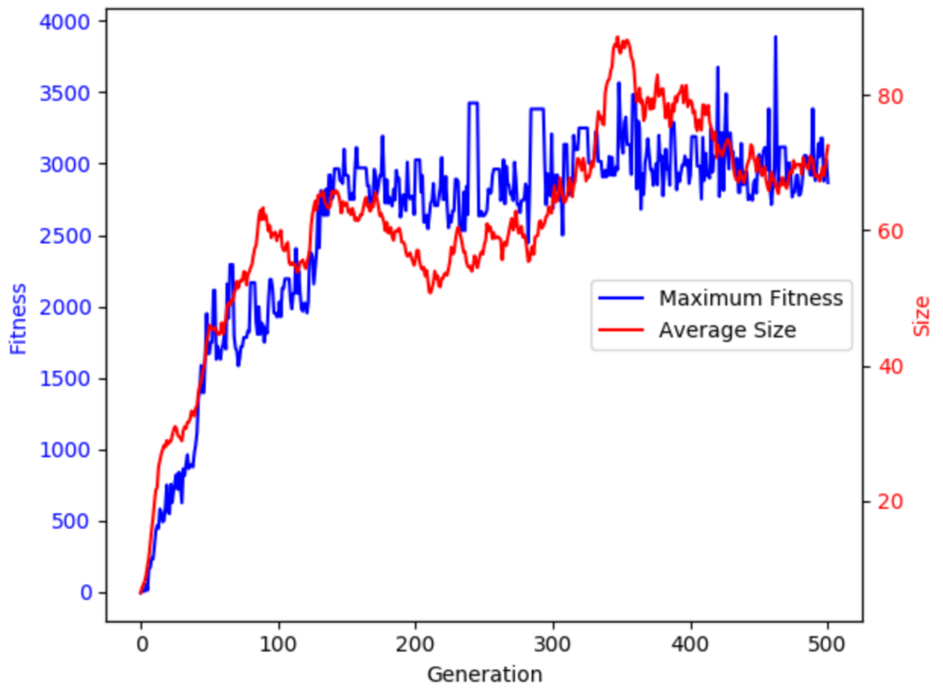


Fig. 10. Results after running 500 generations

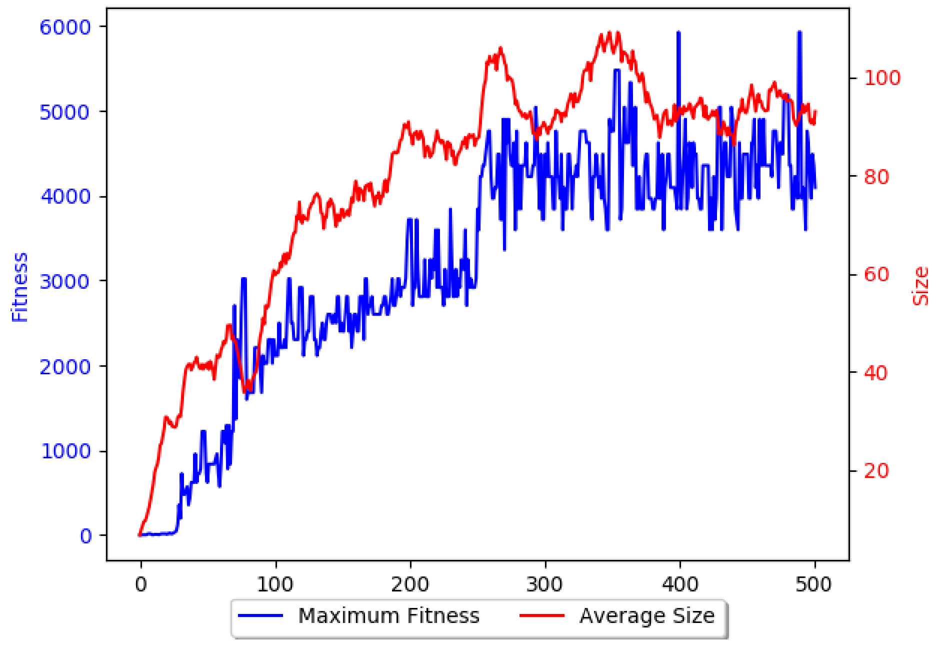


Fig. 11. Results after running 500 generations with the flood-fill implemented

Appendix E
Histograms

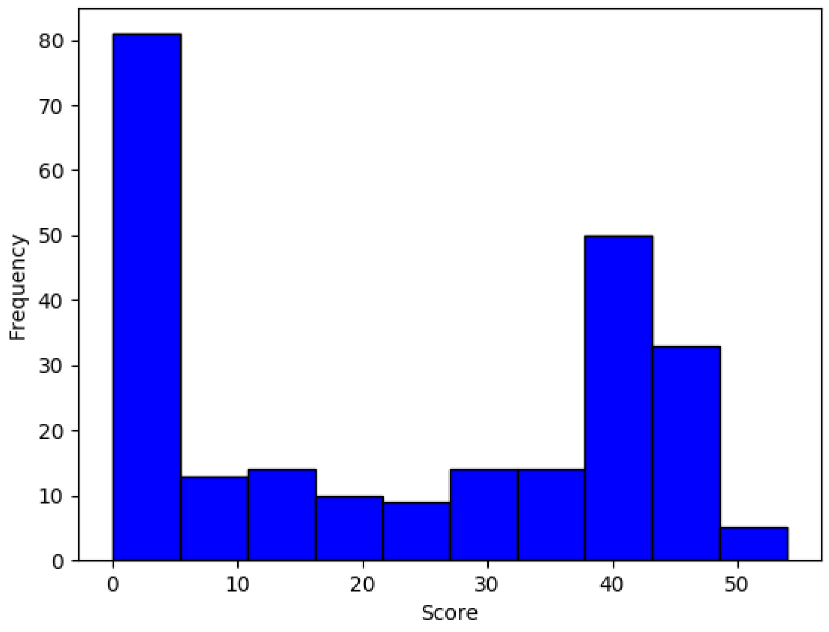


Fig. 12. Histogram of standard function set scores

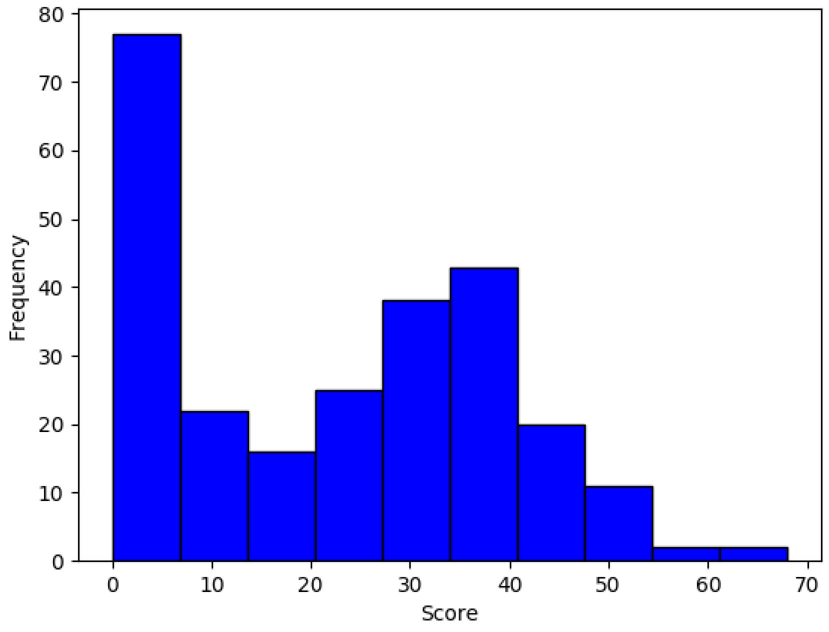


Fig. 13. Score histogram when run using the flood fill algorithm

Appendix F
Function Tree

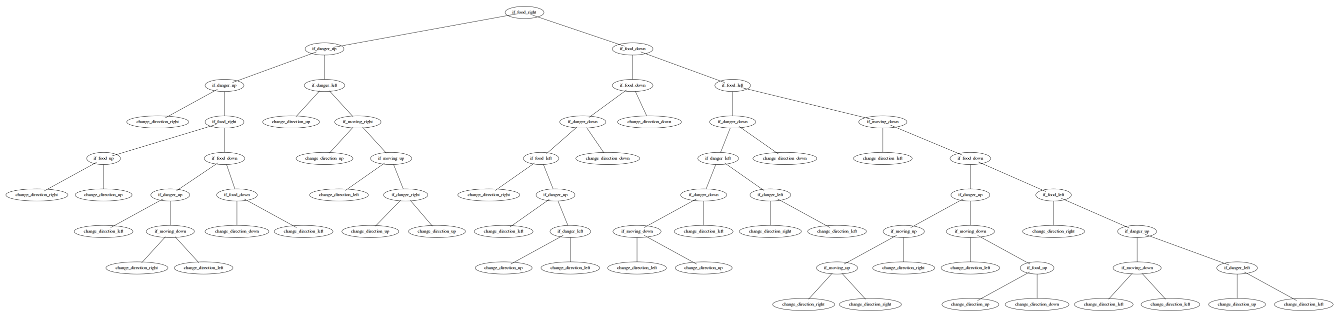


Fig. 14. Output of the best agent’s function tree

End of Examination