MEng/MMath/MSc Degree Examinations 2017/18
DEPARTMENT OF COMPUTER SCIENCE

Evolutionary Computation (EVCO) Individual Assessment

Examination Number: Y1403115

1 Introduction

Snake is a 2D, single player action game. The goal of the game is to navigate a snake through a grid, collecting as many food items as possible. The snake's length grows with each eaten food item. The game is over whenever the snake hits its own body or the wall (in the version of the games where there are walls).

In this report, a genetic programming (GP) approach is presented for evolving an intelligent snake agent, which aims to achieves as high a score as possible. GP has been proven successful in producing intelligent game agents for several games ([1], [2], [3], [4]).

The report is structured as follows: subsection 1.1 presents related work in evolutionary algorithms (EA) for games, as well as heuristic and evolutionary approaches for solving the snake game. section 2 describes the problem and particular approach used to find a solution. section 3 discusses the results obtained from experiments. And section 4 summarises the findings and suggests further work.

1.1 Realated work

In the literature, there are many examples of using evolutionary algorithms to produce intelligent game playing agents. Some of the applications include using EAs to find optimal parameters for game controllers: Pac-Mann [5], The Open Racing Car Simulator (TORCS) video game [6] and Counter Strike [2]; using EA based optimisation to find a game state weighting function: Tetris [7]; using genetic programming to design a controller for Robocode [3]; and using neuroevolution approach to produce agents that play a variety of Atari 2600 games [8].

Even though there is interest in applying EAs to games in general, there is not much research done in using EAs for the snake game. Kong et. al. [9] compares different search algorithms none of which uses EAs. Their findings show the algorithms that perform slower have greater reliability in terms of the snake not crashing before it has eaten all the food.

Yeh et. al. [10] propose a snake controller, which uses a weighted sum of rating functions to determine the best move at each time step. The rating functions are smoothness (number of subsequent moves in the particular direction), space (how many squares can be reached) and distance to food. An EA is used to determine the weights for each rating function. The controllers are evolved in a game environment with fixed food positions. They outperformed a heuristic based approach in the same game environment, However this is not true in a game with random food positions.

Ehlis [1] uses GP to find a routine, consisting of the snakes sensing and movement functions, which achieves the maximum score. This approach is very similar to the solution of the Santa Fe Ant problem described in [4] because it uses the game agent's sensors and actuators directly as a function set in the GA. The paper focuses on finding a function set, which results in the best performance.

2 Methods

2.1 Problem analysis

The game environment is a 14 by 14 grid of squares, with walls in all squares with indexes 0 or 13. The snake starts in square (4,10) and its initial length is 11. On each time step the snake moves one square in the direction it is facing. The snakes action allow it to turn (in place) in one of four direction - up, down, right, left. The directions are defined relative to the grid and not the snake's direction of movement. When the snake ”eats” a food item, a new one is placed at random on a square, which is not occupied by the snake's body. The game ends whenever the snake enters a square, which has a wall or the snake's body in it, or if the snake makes 196 moves without eating any food.

The GP approach used for the snake game was motivated by Ehlis [1] as well as the solution to the Santa Fe Ant problem described in [4] and implemented in [11]. However, there are some subtle differences between these two problems and the snake game. In particular, the turning actions of the snake are defined in terms of the global directions and not the direction, the snake is facing. This means that the snake can potentially turn at 180 degrees, facing its own body and collide with it. This makes the game harder, as there is always a danger in one of the squares adjacent to the snakes head.

It is worth reasoning about which agent behaviour is desirable and which solutions are optimal. A simple optimal solution to the snake game is a Hamiltonian circuit [12] - a path going through all squares of the grid irrespective of where the food is. This way wherever the food appears, the snake will eventually eat it because it is traversing every single square. Ehlis [1] reports that all optimal solutions found follow some Hamiltonian path.

Some important qualities for successful agents include avoiding walls, avoiding own body, avoiding dead ends. Furthermore, if the snake agent is not following a Hamiltonian path it is important for it to aim for the food before the timer runs out.

2.2 Choice of representation

Each individual in the GP algorithm is represented as a tree of operations from a function set. The function set comprises of terminal and non-terminal nodes. The terminal nodes are the movement functions of the snake, which don't have any input parameter or outputs, but just change the direction of the snake. The non-terminals are the sensing functions of the snake and all take as inputs two other functions (terminals or non-terminals) and depending on the evaluation of a certain condition, execute one of them. For example, an individual, who turns right whenever there is food in the square, right of the snake's head, is shown in Figure 2. The tree is equivalent to the python expression if_food_right (changeDirectionRight, changeDirectionUp), which executes changeDirectionRight if there is food in the right square and changeDirectionUp otherwise.

The terminal function set is shown in Table 1.

Two different non-terminal sets were considered. The first one defines food, wall and tail sensing in all four direction relative to the grid. This set will be referred to as the absolute non-terminal set and can be see on Table 3

The second non-terminal set defines the sensing functions relative to the snake's direction of travel. Because the movements are defined in terms of the absolute direction,

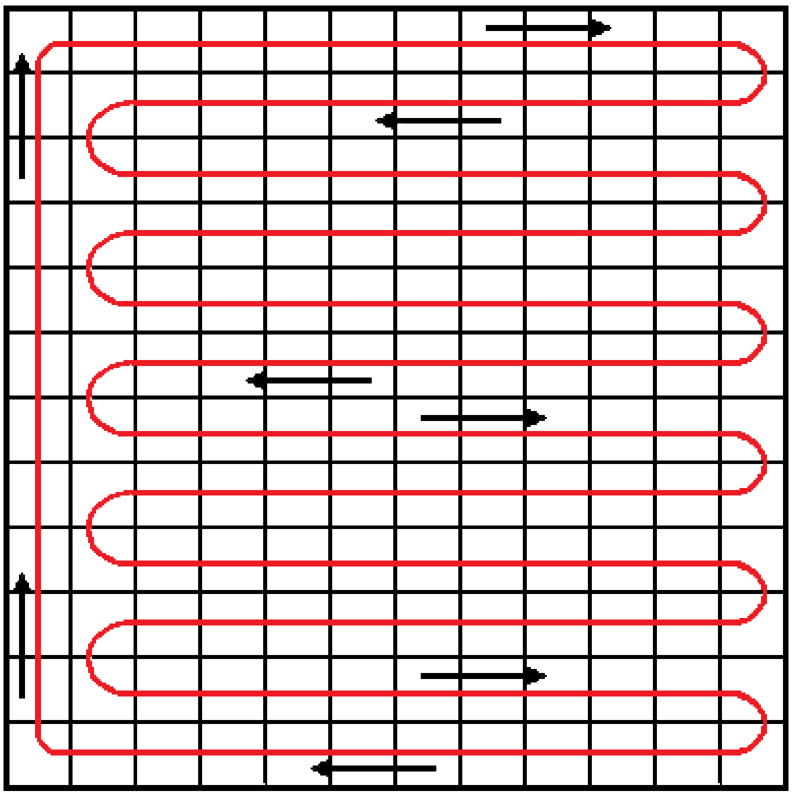


Figure 1: Snake game hamiltonian path

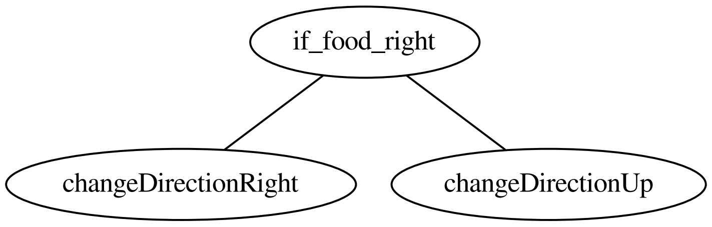


Figure 2: Example tree representation of an individual.

useful routines cannot be constructed. For example, if there is danger on the left of the snake, it does not know in which direction relative to the grid it has to turn to avoid it. For this reason, additional sensing functions are added, which determine the direction in which the snake is travelling. Furthermore, the snake's body will always be behind it so there is no need to include sensing for what is behind. This function set will be referred to as the relative non-terminal set (see Table 2).

This type of representation was chosen because it naturally maps the agents operators to the individual in the GP algorithm. As opposed to using an EA to find some meta parameters for an agent's controller, this approach makes solutions understandable and easier to analyse. It has been successfully applied to the snake game [1] and the Santa Fe Ant problem [4]. A possible downside to this representation might be the

changeDirectionDown, changeDirectionLeft, changeDirectionRight, changeDirectionUp	Changes the direction the snake is facing.
nothing	The snake remains in the same direction.

Table 1: Terminal function set

if_food_up(o1, o2), if_food_down(o1, o2), if_food_left(o1, o2), if_food_right(o1, o2)	If there is food in the line/column of squares to the end of the grid in the specified direction execute first argument otherwise execute second argument
if_wall_up(o1, o2), if_wall_down(o1, o2), if_wall_left(o1, o2), if_wall_right(o1, o2)	If there is a wall in the square to the specified direction execute first argument otherwise execute second argument
if_tail_up(o1, o2), if_tail_down(o1, o2), if_tail_left(o1, o2), if_tail_right(o1, o2)	If there is part of the snake's body in the square to the specified direction execute first argument otherwise execute second argument
prog2(o1, o2)	Executes its first argument, then executes second argument

Table 2: Relative non-terminal function set

if_food_ahead(o1, o2), if_food_on_left(o1, o2), if_food_on_right(o1, o2)	If there is food in the line/column of squares to the end of the grid on the specified side of the snake execute first argument otherwise execute second argument
if_wall_ahead(o1, o2), if_wall_on_left(o1, o2), if_wall_on_right(o1, o2)	If there is a wall in the square on the specified side of the snake execute first argument otherwise execute second argument
if_tail_ahead(o1, o2), if_tail_on_left(o1, o2), if_tail_on_right(o1, o2)	If there is part of the snake's body in the square on the specified side of the snake execute first argument otherwise execute second argument
if_moving_up(o1, o2), if_moving_down(o1, o2), if_moving_left(o1, o2), if_moving_right(o1, o2)	If the snake is moving in the specified direction execute first argument otherwise execute second argument
prog2(o1, o2)	Executes its first argument, then executes second argument

Table 3: Absolute non-terminal function set

large function set. A large function set might require a substantial amount of computational resources to find a solution if the fitness landscape is rough.

2.3 Fitness evaluation

Two different fitness functions were considered and experimented with. The first one (see Equation 1) is just the number of food items the snake eats during a single run (the score).

$$Fitness = score \tag{1}$$

The second fitness function considered, is the sum of the score and the total number of moves the snake has made during the run. In addition, the snake's fitness is penalised if it has died because it has not eaten any food for 196 moves. This fitness function can be seen in Equation 2. where $steps$ is the total number of moves the snake has made, $timer = 196$ and $I(timerExpired)$ is a boolean function such that $I(timerExpired) = 1$ if $timerExpired = True$ and $I(timerExpired) = 0$ otherwise. $TimerExperied$ is true if the snake has made 196 moves without eating any food.

$$Fitness = score + steps - I(timerExpired) * timer \tag{2}$$

The choice of the second evaluation function was motivated by the fact that the Hamiltonian Figure 1 path solutions do not explicitly aim for the food. A possible way to motivate individuals to tend towards such a solution, is to give the higher fitness for traversing the grid. Nevertheless, there has to be a trade-off between the pursue of food and the number of moves made. In earlier generations individuals would tend to go in a loop in order to maximise the $steps$. In order to ensure this does not happen, the penalty is introduced.

2.4 Evolution operators and parameters

In Table 4 the evolution operators and parameters are given. They either arise from the problem domain or were found empirically. Population size and number of generations were highly restricted by the computational resources available, but were set to as high values as possible.

2.4.1 Initialisation

For the initialisation of individuals the ramped half and half method [4] is used. It was chosen in order to increase diversity in the initial population. The maximum depth was determined empirically. There cannot be a useful tree with depth one, so the minimum was set to 2. Although, in most cases initialisation produces a good initial population, in some cases bloat occurs, even with bloat control in place (see subsubsection 2.4.5), because of the high maximum tree depth. However, it had to be set to a high enough value, in order to encourage diversity.

2.4.2 Selection

Initially a simple selection tournament was used as the selection operator. However, in order to limit bloat, a selection double tournament was chosen later on. Tournament size was chosen empirically and based on computational resources available. The parsimony size was set to relatively small value, in order not to discourage diversity.

Population size	600
Number of generations	400
Crossover rate	0.6
Mutation rate	0.1
Initialisation	Ramped half and half
Initialise min depth	2
Initialise max depth	9
Selection	Selection double tournament
Tournament size	7
Parsimony size	1.2
Crossover	One point crossover with leaf bias
Leaf crossover probability	0.1
Mutation	Uniform
Mutation tree initialisation	Full
Mutation Min Depth	1
Mutation Max Depth	2
Bloat control	Static limit on number tree nodes
Max Nodes	150
Hall of fame size	1

Table 4: Evolution operators and parameters

2.4.3 Crossover

The crossover operator was chosen based on the specific problem - applying crossover to the leaf nodes in the tree essentially replaces one movement action with another. This can rarely lead to beneficial constructs in the offspring, but can lead to good solutions being destroyed. For this reason, crossover with a leaf node bias was used. The probability of choosing a leaf as a crossover point was set 0.1 and the probability of choosing a none leaf node - 0.9. The overall crossover rate was chosen empirically to be 0.6.

2.4.4 Mutation

The chosen mutation operator is uniform mutation and uses the full method to construct new trees. In this particular problem, mutations can often ruin existing solutions, rather than finding new ones. Furthermore, it is unlikely that randomly generated trees of great depth would yield beneficial constructs. For this reason, the maximum depth of generated trees was set to 2 and the overall mutation probability of the algorithm to 0.1.

2.4.5 Bloat control

In order to mitigate the effects of bloat, a static limit on the number of nodes in each individual tree was used. This limit was applied to mutation and crossover, so any individual produced by the two operators, which had a grater number of nodes, was discarded. Limiting tree size was also necessary due to limited computation resources. There is also evidence that for the similar Santa Fe Ant problem, the proportion of solutions, in smaller trees, is higher [13].

Limiting tree depth was also considered. However sparse deep trees were observed to have a high fitness in some cases. Also it is not clear what the depth limit should be set to without limiting the solution space.

Although, the size limit is applied to crossover and mutation, it is not applied to initialisation, meaning that bloat can still occur. However, it is much more unlikely to occur, because of the size limit and the double selection tournament, which favours smaller individuals.

3 Results

Three versions of the algorithm were compared to determine the best one. They use a different combination of the function sets and evaluation functions, presented in section 2. The specification for each algorithm is shown in Table 5.

The algorithm evaluation procedure is described in subsection 3.1. The results of each algorithm are presented and discussed in subsection 3.2.

Algorithm	**Evaluation Function**	**Function set**	**parameters**
Algorithm 1	Composite (Equation 2)	Absolute non-terminals (Table 3) and terminals (Table 1)	from Table 4
Algorithm 2	Simple (Equation 1)	Absolute non-terminals (Table 3) and terminals (Table 1)	from Table 4
Algorithm 3	Composite (Equation 2)	Relative non-terminals (Table 2) and terminals (Table 1)	from Table 4

Table 5: Algorithms

3.1 Algorithm evaluation

Different algorithm were compared using their score, as opposed to using their fitness. This decision was made because algorithm 2 uses a different evaluation function from the others and also the aim of this study is to find an individual that achieves a maximum possible score.

Each algorithm was ran 20 times. The best individual for each run was recorded and evaluated 100 times on 100 test problems. In order to perform a fair comparison between algorithms, for each test problem (from 0 to 99), the random seed was fixed to the problem number(from 0 to 99). This means that the food in the game environment was placed in the same positions for each run of the same test problem. For each of the best individuals, an average score was computed from all 100 test problems.

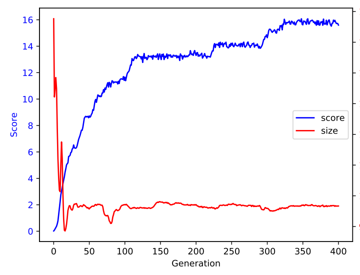
3.2 Experimental results

On Figure 3 the average score and size for each algorithm is shown. Comparing algorithm 1 (Figure 3a) with algorithms 2 (Figure 3b) and 3 Figure 3c, it achieves the highest score. Furthermore, the average size of algorithm 1's tree is the smallest of the three.

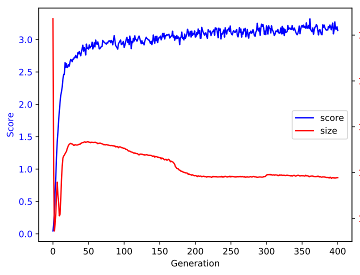
This relationship between score and size, could be an indication that the proportion of good solution is higher in smaller tree sizes, as suggested by [13].

Algorithm 2 converges the fastest out of the three and it's average size starts off close to the limit of 150 in early generations and goes down in later generations. The quick convergence can be accounted to the simpler fitness function. Agents quickly learn to pursue the food in earlier generations, but struggle to learn to avoid danger and maximise their number of moves in later generations.

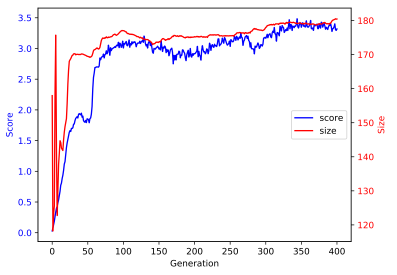
Algorithm three achieves similar results to algorithm 2 in terms of the score achieved, however the average tree size is much larger than any of the others. In fact the size is above the size limit of 150, which indicates that bloat has occurred in several runs. This may be due to the fact that beneficial constructs in the tree, require combinations of more nodes, since the direction of travel has to be determined before each move can be made.



(a) Algorithm 1



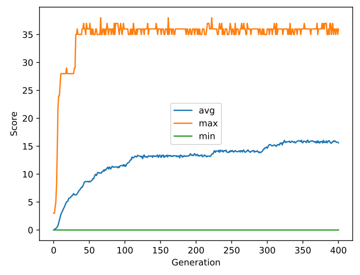
(b) Algorithm 2



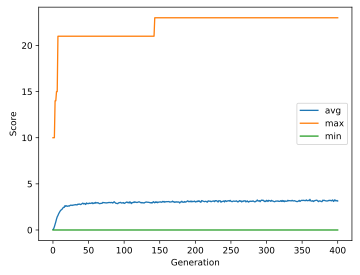
(c) Algorithm3

Figure 3: Average score and average size for each generation out of 20 runs.

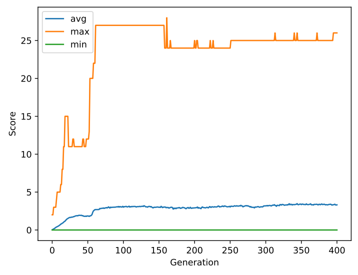
In figures Figure 4a, Figure 4b and Figure 4c the maximum and minimum scores are shown in addition to the averages. The minimum score for all three algorithms is 0, which is to be expected, as mutation and crossover can destroy good individuals. The maximum scores achieved however, are much closer to each other. Algorithm 1 achieves the best maximum score, followed by 3 and 2. It is worth noting that, despite of bloat, algorithm 3 performs better than 2 in terms of maximum score and has a similar performance in terms of average score. This, perhaps, indicates the importance of the evaluation function.



(a) Algorithm 1



(b) Algorithm 2



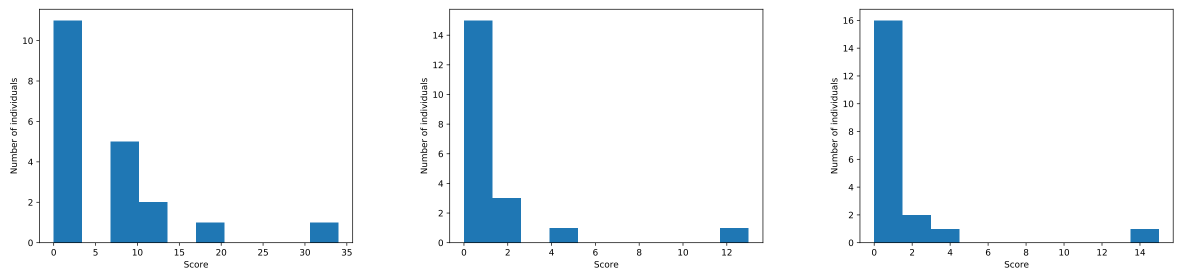
(c) Algorithm3

Figure 4: Average maximum and minimum score for each generation out of 20 runs.

Figures Figure 5a, Figure 5b and Figure 5c show the performance of the best 20 individuals on the test problems. Looking at the distributions, we see that for each algorithm there is a single best individual, which is an outlier. The distributions of algorithms 2 and 3 are very skewed and look almost identical. On the other hand, the distribution of algorithm 1 is more spread out. Furthermore, almost half of the individuals from algorithm 1 seem to outperform the individuals from algorithms 2 and 3, which is strong evidence that algorithm 1 is indeed better.

However, GP algorithms are influenced by many random factors and can converge to different solutions across different runs. For this reason statistical tests should be used ([14], [15]), in order to determine if there is a significant difference between algorithms.

A Mann–Whitney U test was conducted, in order to compare each pair of algorithms. The results of the test for each can be seen in Table 6. The null hypothesis of the Mann–Whitney U test is that the two tested samples come from the same distribution. It is rejected if the resulting p-value is less than 0,05. The p-values, obtained for algorithms 1 and, and algorithms 1 and 3 are below the threshold of 0.05. Therefore, it is safe to assume that algorithm 1 is better than the other two. In contrast, the p-value, from the test conducted on algorithms 2 and 3, is higher 0.05, which means that it is unclear weather or not one is better than the other.



(a) Algorithm 1 (b) Algorithm 2 (c) Algorithm3

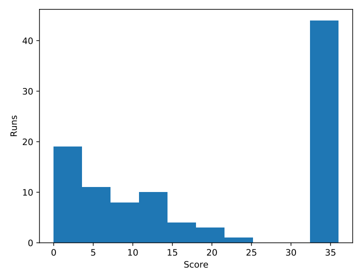
Figure 5: Distribution of individuals evaluated on 100 test runs over their average score.

First algorithm	Second algorithm	p-value
Algorithm 1	Algorithm 2	0.0038075456551803864
Algorithm 1	Algorithm 3	0.0010537832718209305
Algorithm 2	Algorithm 3	0.15610108279449836

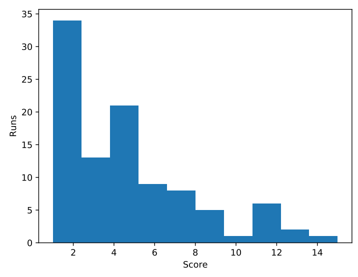
Table 6: Mann–Whitney U test results

Figures Figure 6a, Figure 6b and Figure 6c show the scores of the single best individuals evaluated on the 100 test problems. For about 50% of the runs, algorithm 1 has achieved a score of around 35. However for nearly 20 of the runs the score is around 0. Algorithm 2 has a very skewed distribution =, with only a few test problem evaluations with a score above 10 Algorithm 3 again has a skewed distribution, however the spread is much smaller with all test runs evaluating between 14 and 17.

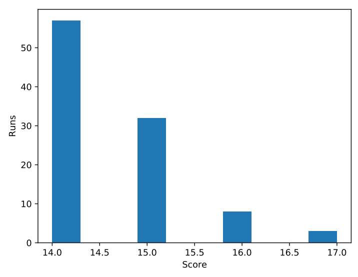
The strategy of almost all individuals, achieving a score above 25, was to stick next to wall, circling the grid, and going away from the wall only to get food. They would also learn to sense food only in one direction. Some of them managed to evolve, to avoid their own body in certain situations, however none of them managed to avoid it in all situations. This might suggest that given more generations they could evolve this essential strategy. None of the individuals followed a Hamiltonian path.



(a) Algorithm 1



(b) Algorithm 2



(c) Algorithm3

Figure 6: Distribution of results for 100 evaluations of the single best individual.

4 Conclusion

A genetic programming algorithm was developed in order to evolve an intelligent agent to play the snake game. The effects of different evaluation functions and different non-terminal sets on the convergence, score and tree size of the algorithm were compared. It was shown that a multi objective evaluation function, with a penalty for individuals who do not seek food, results in slightly slower convergence, but a much higher score. The choice of non-terminal function set, which allows the construction of smaller beneficial sub trees, resulted in a higher score. The algorithm that performed best was shown to maintain a smaller tree size compared to the others, which might be due to the relationship between density of high fitness individuals and tree size [13].

Out of the identified beneficial behaviour, agents managed to learn to avoid walls and seek food, but not to avoid their own body. None of the solutions managed to evolve a strategy to follow a Hamiltonian path and none of them managed to achieve the maximum score.

4.1 Future Work

The algorithm was evaluated using two different non-terminal sets. In future work it might be worth, trying different combinations of the two sets.

The performed experiment were highly restricted by computational resources. Ehlis [1] used a population of 10000 for a similar version of the same problem and managed to achieve the maximum score. This suggests that it might be worth evaluating the algorithm, presented in this work, with a larger population size.

Langdon et. al. [13] criticise GP algorithms on the lack of ability to exploit inherent symmetries in the solution space. They show that the performance of GP, on the Santa Fe Ant problem, is slightly better than that of random search. The representation of individuals they use is the same as the one used in this work. This suggests that using a different representation such as the one in [10] might yield better results.

The approach presented in [16] systematically considers all small trees in the search space of GP. Adapting it to the snake game problem would potentially speed up convergence significantly.

References

[1] T. Ehlis, *Application of Genetic Programming to the Snake Game*, en-US, 2000. [Online]. Available: https://www.gamedev.net/articles/programming/artificial-intelligence/application-of-genetic-programming-to-the-snake-r1175/ (visited on 01/13/2018).

[2] N. Cole, S. J. Louis, and C. Miles, “Using a genetic algorithm to tune first-person shooter bots,” in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, vol. 1, Jun. 2004, 139–145 Vol.1. doi: 10.1109/CEC.2004.1330849.

[3] Y. Shichel, E. Ziserman, and M. Sipper, “GP-Robocode: Using Genetic Programming to Evolve Robocode Players,” en, in *Genetic Programming*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Mar. 2005, pp. 143–154, isbn: 978-3-540-25436-2 978-3-540-31989-4. doi: 10.1007/978-3-540-31989-4_13. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-31989-4_13 (visited on 01/21/2018).

[4] J. R. Koza, *Genetic programming: On the programming of computers by means of natural selection*, ser. Complex adaptive systems. Cambridge, Mass: MIT Press, 1992, isbn: 978-0-262-11170-6.

[5] M. Gallagher and A. Ryan, “Learning to play Pac-Man: An evolutionary, rule-based approach,” in *The 2003 Congress on Evolutionary Computation, 2003. CEC ’03*, vol. 4, Dec. 2003, 2462–2469 Vol.4. doi: 10.1109/CEC.2003.1299397.

[6] J. Muñoz, G. Gutierrez, and A. Sanchis, “Multi-objective evolution for Car Setup Optimization,” in *2010 UK Workshop on Computational Intelligence (UKCI)*, Sep. 2010, pp. 1–5. doi: 10.1109/UKCI.2010.5625607.

[7] N. Böhm, G. Kókai, and S. Mandl, *An Evolutionary Approach to Tetris*. Jan. 2018.

[8] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, “A Neuroevolution Approach to General Atari Game Playing,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 355–366, Dec. 2014, issn: 1943-068X. doi: 10.1109/TCIAIG.2013.2294713.

[9] S. Kong and J. A. Mayans, “Automated Snake Game Solvers via AI Search Algorithms,” University of California, Irvine, Irvine, California, USA, Tech. Rep. [Online]. Available: http://sites.uci.edu/joana1/files/2016/12/AutomatedSnakeGameSolvers.pdf (visited on 01/21/2018).

[10] J. F. Yeh, P. H. Su, S. H. Huang, and T. C. Chiang, “Snake game AI: Movement rating functions and evolutionary algorithm-based optimization,” in *2016 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, Nov. 2016, pp. 256–261. doi: 10.1109/TAAI.2016.7880166.

[11] *Artificial Ant Problem — DEAP 1.2.2 documentation*. [Online]. Available: http://deap.readthedocs.io/en/master/examples/gp_ant.html (visited on 01/21/2018).

[12] *Hamiltonian path*, en, Page Version ID: 814872829, Dec. 2017. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Hamiltonian_path&oldid=814872829 (visited on 01/21/2018).

[13] W. B. Langdon and R. Poli, “Why ants are hard,” 1998.

[14] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” in *Software Engineering (ICSE), 2011 33rd International Conference on*, IEEE, 2011, pp. 1–10.

[15] J. Derrac, S. García, D. Molina, and F. Herrera, “A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms,” *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 3–18, Mar. 2011, issn: 2210-6502. doi: 10.1016/j.swevo.2011.02.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210650211000034 (visited on 01/22/2018).

[16] S. Christensen and F. Oppacher, “Solving the artificial ant on the Santa Fe trail problem in 20,696 fitness evaluations,” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ACM, 2007, pp. 1574–1579.