Evolutionary Computation

Y1481702

January 22, 2018

Contents

1 Abstract

2 Introduction

2.1 Background

Biomimicry, the imitation of nature for solving human problems has produced many examples of world class design such as Velcro(R), self-cleaning paints and Shinkansen high-speed trains[1]. Since the term biomimicry was first coined in 1997 by J. Benyus, this design philosophy has started to inspire many new developments. As well as the aforementioned mimics of physical systems in nature, natural processes have also been imitated, such as in the circular economy in product design and swarm robotics in computer science.

Evolutionary Computation (EC) is a machine learning technique which is directly inspired by biological evolution. This is clearly another example of an imitation of nature's processes. Life has been around on earth for over 3.5 billion years, this is a lot of research & development time to produce useful solutions which can be related to many of the current problems in a wide range of fields. Through evolution, nature has already produced its own solutions to many of the hardest machine learning problems of today such as vision.

Machine learning usually focuses on optimisation of a particular goal, or set of goals, in a similar way that human engineers aim to produce optimal designs. Benyus sums this up well: 'Computers can generate random ideas much faster than most engineers. And computers, not yet able to feel embarassment or peer pressure, are not afraid to try off-the-wall ideas. Ideas are just ideas; the more the merrier.'[1, pp.209]

2.2 DEAP

Distributed **E**volutionary **A**lgorithms in **P**ython (DEAP) is an open source framework for Python[2]. The framework aims to provide tools for quickly producing custom evolutionary algorithms. It provides a large amount of in-built functionality for implementing commonly used evolutionary algorithms and allows for parts of these to be mixed and matched as well as intermixed with custom sections.

2.3 EC for Snake

The task given is to create an evolutionary algorithm in Python using the DEAP framework in order to play the classic video game, Snake. This is not a new task and has been the subject of previous work including implementations using genetic programming[3] and more recently genetic algorithms[4]. Despite this there is still room for further work, due to ongoing research into evolutionary computing and significant challenges because of the nature of the game.

There are a variety of available instances of Snake with an array of different rules. In this case, there is a square game board of 14x14 cells as shown in Figure. 1. The snake will continously move forwards in the direction it is facing, which can be changed by the user at any time. The aim is to collect food that appears in a random cell on the board when the previous one is eaten. Each time the snake eats a piece of food, it grows and takes up an extra cell. Once the snake fills every cell it can no longer grow. Hence, the highest possible score (185) is equal to the number of cells (196)

less the initial length of the snake (11).

Implementing this task will involve facing all of the usual challenges in evolutionary computation such as maintaining diversity and reducing bloat. Snake also brings along new challenges. The random placement of the food on the board adds an element of stochasticity to the solution meaning algorithm will perform differently across multiple runs and care will need to be taken to ensure that it works in the general case. The solution will also need to balance the need for the snake to avoid crashing into walls or itself while still picking up the food before time runs out. Optimising both of these objectives at the same time might be difficult.

Starter sensing functions are given that tell the snake if there is a wall, itself or food directly ahead

3 Methods

3.1 Technology

All the described algorithms have been written in Python 2.7.12 and DEAP Version 1.2. They have been run using standard University of York Computer Science lab PCs running Ubuntu 16.04 and containing Quad-Core Intel Core i7-4770 and 16GB of RAM.

3.2 Representation

3.2.1 Available Options

The representation links the original problem to the search space over which the algorithm runs. As such, the type of representation chosen can significantly affect the size of the search space and by extension, the time needed to search it.

Genetic Algorithms (GA)
GAs are perhaps the most commonly used form of evolutionary algorithm. They are generally used for optimisation problems but have been applied to a variant of the snake game before[4]. In this case, the GA was used to optimise four parameters (smoothness, space, food and direction) which determined the snake's movement.

Evolutionary Strategy (ES)

Genetic Programming (GP)
The aim of GP is to evolve programs that can be used to solve a problem. Clearly this is immediately more relevant for our snake implementation, the aim would be to create a program that can be run to determine our next move given the current state. GP uses a tree representation for programs, where the internal nodes are functions and the leaf nodes of the tree are constants and variables.
GP has previously been explored by Ehlis[3] and produced some promising results. Different function sets have been explored as well as a technique called priming which... [what is this?].

Neuroevolution
Neuroevolution is a hybrid technique between evolutionary computing and artificial neural networks (ANN). Unfortunately as the brief requires that the DEAP Python library is used, neuroevolution is not an available option as DEAP does not support it. Grammatical Evolution? Is this explicitly supported in DEAP?[2] DEAP's transparency allows virtually any algorithm to be implemented but it's easier to use the built in structures.

3.2.2 Choice and Justification

Genetic Programming will be used as it most closely fits the aims set out in the introduction above. A previous implementation of GP for snake is available online[3], this has been re-written in DEAP and slightly adapted to allow it to work with the available controls (up, down, left, right). Originally, this code was run with a population of 10,000 across 500 generations. Due to time constraints, this amount of computing was not feasible and needed to be considerably cut down. The final results have been generated using a population of X across Y generations. While the highly successful results from this original experiment could not be full reproduced, this new implementation gives a good baseline for judging further improvements. Any changes proposed in this report will be compared against this baseline in order to produce meaningful results. In order to avoid anomalous results caused by randomness, all results given are the mean result over 30 code runs.Several sensing functions have been given in the problem definition which all return boolean.

3.2.3 Physical Environments

This application of GP fits into a specific category involving a physical enviroment, which in this case is simulated. The added difficulty comes from the fact that the execution of any terminal element may change the environment. Once this change has been made, the subsequent execution may produce a different effect in the environment. In this kind of application the internal EA mechanics remains the same, but fitness evalutation can be significantly more computationally expensive[5, p. 110] in order to get a fair representation. In this example, as with the control, the fitness of a particular program will need to be determined over a full game, instead of a single turn. This will be discussed further in Section 3.4.

Previous work has shown that the function set can help to find a better solution, however we need to be careful of overparameterising the problem.This is a common trade-off in many forms of machine learning? The control code has an expanded function set from the original brief 2.

3.2.4 Initialisation Procedure

DEAP offers all of the three main initialisation methods used in genetic programming. The Full method initialises a program tree where every individual in the population is the same depth along every path. The grow method creates trees with nodes that terminate before the maximum height is reached. The final type of initialisation procedure, ramped half and half, creates half of the trees using the full method and half using the grow method. This is what will be used for this algorithm as it will provides the greatest diversity in initial population.

3.3 Population

3.3.1 Size

Having the correct population size is crucial to success[6]. The size of the population needs to be large enough to explore a significant amount of the search space by containing a number of promising individuals. If this is not the case, a large amount of mutation will be required to find any good solutions creating a very unstable solution. Too large a population will significantly impact the algorithm's performance. As the population size tends towards the size of the search space, the algorithm becomes no better than a brute force solution, running unnecessarily slowly. It has also been mathematically proved that, for certain problems, a larger population is highly likely lead to a sub-optimal solution if the problem has an attraction basin near some local optimum[7].

In almost all EA applications the population size is constant[5, p. 20]

The algorithm was tested with varying population sizes to show the affect of this on the runtime and the fitness of the final individual produced. The algorithm was run 30 times for each population size, the runtime and fitness values for this are shown in Table 1 and by Figure 2a. After each run, the most fit individual was used to play 500 independent game, these results are shown in Table 1 and by Figure 2b. The runtime values only include the time used for evolution, not for testing the final individual.

It is worth noting that the control has a mutation value of 0, so it requires that the initial population contains a reasonable distribution of good individuals. Adding mutation would allow novel ideas to be introduced later on in the evolution. This is likely why larger populations perform significantly better with this control algorithm and will be explored later in Section 3.6.1. Another interesting feature in the data is the significant increase in standard deviation as the population size increases. This shows that while larger populations help to produce individuals that perform better overall, this performance is not consistently across multiple games. This issue will be important to address later.

The optimal size for the population is the size where a good solution is often produced but the program is quick to converge[8]. A population of 1,000 seems to give a solution that is reasonably quick to converge (approx. 50 generations, Fig. 2a) but also produces a reasonable score across multiple games (around 6.97, 1). This size of population will mean that proposed changes to the algorithm in this paper can be tested in a reasonable amount of time even with the limited timespan for the project and available computing resources.

3.3.2 Diversity

Within this evolutionary computation task, each individual of the population corresponds to a point in the search space of all possible programs. Diversity corresponds to having these points reasonably spread out throughout this space. A diverse population is, by definition, exploring more of this search space than one that is not.

3.4 Evaluation

Implementing a good fitness evaluation function is essential to evolving a solution to this problem. Without this, there will be no way to assess individuals in order to determine which will survive and

reproduce. There are two major objectives at play in the game, avoiding crashing the snake and getting to the food before the timer hits zero. These can be represented by the final size of the snake when the game ends, as if the snake crashes then the game ends and it is unable to get to the food and grow. Therefore, we can simply consider this a single-objective problem where the objective is produce the maximum length of snake before either the snake crashes or the timer hits zero. This significantly simplifies the problem with minimal impact.

The snake's food spawns randomly around the board. This element of stochasticity makes evaluating the fitness of a particular solution more difficult. The fitness of a given algorithm will likely change between games and this has already shown by previous results, as mentioned in Section 3.3.1. In order to ensure a fairer representation of fitness, each solution may need to be evaluated across multiple games. A number of different fitness functions were tested. Each of these have different benefits - factorial (single run), increasingly rewards higher scores - factorial (multiple runs), heavily rewards individuals which have a consistent high score

3.5 Parent Selection

Selecting which individuals should have influence over the next generation is one of the most important factors for success in Evolutionary Computing. If only the best individuals are selected then diversity will be lost, the algorithm will get stuck in a local minima and be unable to find a good general solution. This is known as premature convergence. If too many of the less-promising solutions are selected then the algorithm may be unstable meaning good solutions will take too long to find and may be lost quickly. Striking a good balance between a diverse population and a stable algorithm is necessary for success in this task.
In biological evolution, old generations die to free up resources for newer generations. Disregarding solutions due to age ...
Parent Selection can be random, so (if this is the case) good solutions may well die out. Elitism can be used to ensure that the best solution isn't randomly removed from the population. If no better solution is ever found, this solution will want to be present in the final population once evolution is complete.

3.5.1 Maintaining Diversity

Diversity is difficult to quantity in an objective manner and, as such, no single measure for it exists. A range of different measures including the number of different fitness values, number of different phenotypes/genotypes or entropy may be used[5].

Roulette Wheel

Tournament

3.5.2 Bloat

Bloat, the increase of the size of a genetic program without significant increase in fitness, is a difficult problem to overcome. Bloat is linked to, but different from, overfitting, a common problem in many forms of machine learning[9].

Controlling bloat while maximising fitness turns the evolution into a multi-objective optimisation problem. The two most commonly used solutions to bloat are parsimony pressure and double tournaments. Parsimony pressure subtracts a value based on the size of the program tree from the individual's fitness, this value does not necessarily need to be linear. Double tournaments can be used for parent selection as an extension of regular tournament selection described in Section 3.5.1. It has been shown that the order of a double tournament has no significant bearing on the end result. [cite, practical?]

3.6 Variation Operators

3.6.1 Mutation

Mutation is the operator which incorporates new ideas into the population. Without it, genes would only be shuffled around and no new solutions could be found. A trade-off is required here, too much mutation will result in a highly unstable algorithm. Far too much mutation would Care needs to be taken when implementing mutation so as not to create biases. This is particularly the case with real-valued mutation..

3.6.2 Recombination

3.6.3 Termination Condition

Certain number of generations..

4 Results

5 Conclusion

5.1 Main Findings

Significant

5.2 Further Work

The complexity and diversity of the snake game can lead into plenty of areas for further exploration. Recent developments in evolutionary computation such as co-evolution and neuroevolution could be explored in the context of this task. With co-evolution individuals would have to compete in order to get to the food items first, creating snake players that find the food faster.

A snake that cycles through each square will always each piece of food as it traverses. An alternative implementation of this algorithm could have a function set with no information of the position of the food, but reward the snake using a fitness function based on the number of unique squares visited before returning to the start position. This alternative algorithm could be compared to the one described in this report to determine which approach is best.

Future work could also explore the adaptability of this program to perform well across the wide number of variants of the game that exist with different formats and rules. This could be further expanded to attempt to create individuals that generalise across multiple game variants and carry their learning across in the way a human player would.

6 References

[1] J. M. Benyus, *Biomimicry: Innovation Inspired by Nature.* HarperCollins, May. 1997.

[2] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, Jul. 2012.

[3] T. Ehlis. (2000, Aug) Application of genetic programming to the snake game. [Online]. Available: https://www.gamedev.net/articles/programming/artificial-intelligence/application-of-genetic-programming-to-the-snake-r1175/ [Accessed: Nov. 30, 2017]

[4] J.-F. Yeh, P.-H. Sun, S.-H. Huang, and T.-C. Chiang, "Snake game ai: Movement rating based functions and evolutionary algorithm-based optimization," in *2016 Conference on Technologies and Applications of Artificial Intelligence, Proceedings*, Nov. 2016.

[5] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing.* Springer, 2007.

[6] E. Spinosa and A. Pozo, "Controlling the population size in genetic programming," in *Advances in Artificial Intelligence*, G. Bittencourt and G. L. Ramalho, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 345–354.

[7] T. Chen, K. Tang, G. Chen, and X. Yao, "A large population size can be unhelpful in evolutionary algorithms," *CoRR*, vol. abs/1208.2345, 2012. [Online]. Available: http://arxiv.org/abs/1208.2345

[8] S. Rylander and B. Gotshall, "Optimal population size and the genetic algorithm," *Population*, vol. 100, no. 400, p. 900, 2002.

[9] L. Vanneschi, M. Castelli, and S. Silva, "Measuring bloat, overfitting and functional complexity in genetic programming," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 877–884. [Online]. Available: http://doi.acm.org/10.1145/1830483.1830643

[10] S. Luke and L. Panait, "Fighting bloat with nonparametric parsimony pressure," *Parallel Problem Solving from Nature*, pp. 411–421, 2002.

[11] E. D. de Jong, R. A. Watson, and J. B. Pollack, "Reducing bloat and promoting diversity using multi-objective methods," in *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 11–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=2955239.2955241

7 Appendix

Population Size	Run Time		Mean Test Run Score	
	Mean	6.59	Mean	2.76
100	Max	13.5	Max	11.60
	St.Dev	2.52	St.Dev	2.14
	Mean	17.7	Mean	4.67
250	Max	28.9	Max	18.8
	St.Dev	4.22	St.Dev	3.48
	Mean	43.39	Mean	5.81
500	Max	121.75	Max	21.97
	St.Dev	18.55	St.Dev	3.99
	Mean	66.5	Mean	7.31
750	Max	252.96	Max	23.9
	St.Dev	38.66	St.Dev	4.46
	Mean	78.59	Mean	6.97
1000	Max	133.16	Max	24.20
	St.Dev	21.63	St.Dev	4.61
	Mean	249.24	Mean	10.73
2,500	Max	518.06	Max	29.40
	St.Dev	83.79	St.Dev	5.73
	Mean	572.78	Mean	13.99
5,000	Max	997.97	Max	33.83
	St.Dev	185.05	St.Dev	6.64
	Mean	1437.17	Mean	18.0
10000	Max	39.27	Max	39.3
	St.Dev	7.59	St.Dev	7.59

Table 1: Population Size Performance

Evaluation	**Description**	**Test Run Score**	
Score	description here	Mean Max St.Dev	X Y Z
Score Squared	description here	Mean Max St.Dev	X Y Z
Score Factorial	description here	Mean Max St.Dev	X Y Z

Table 2: Fitness Evaluation Performance

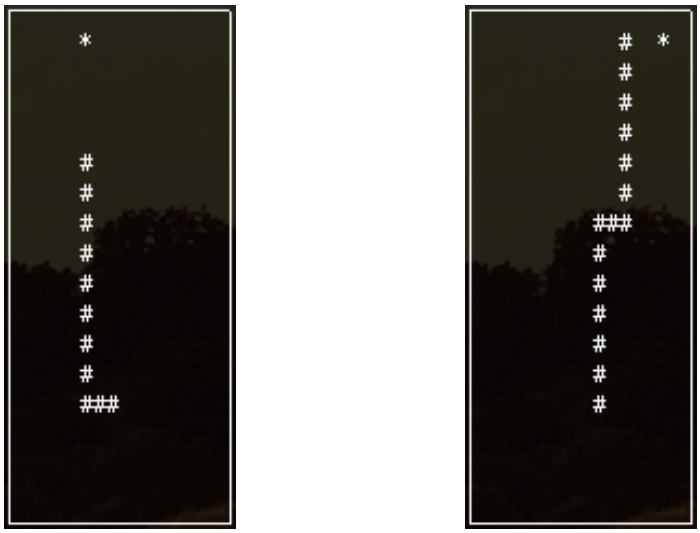
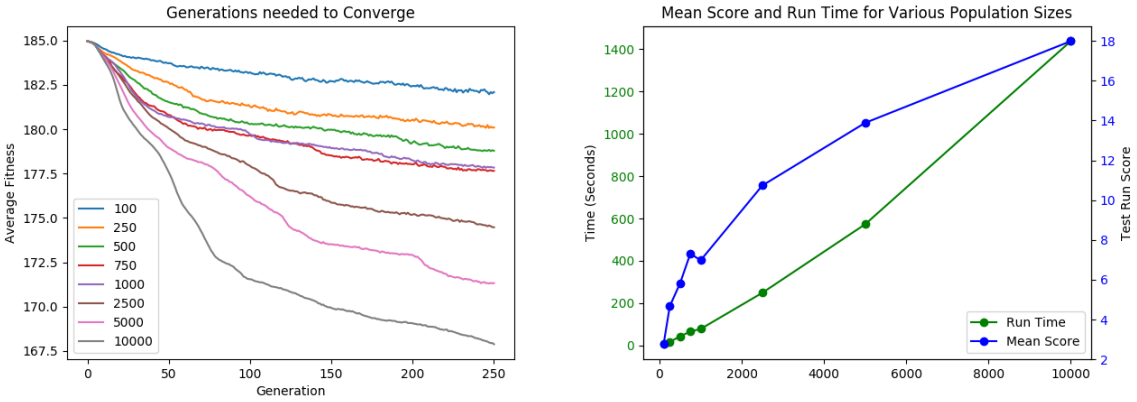


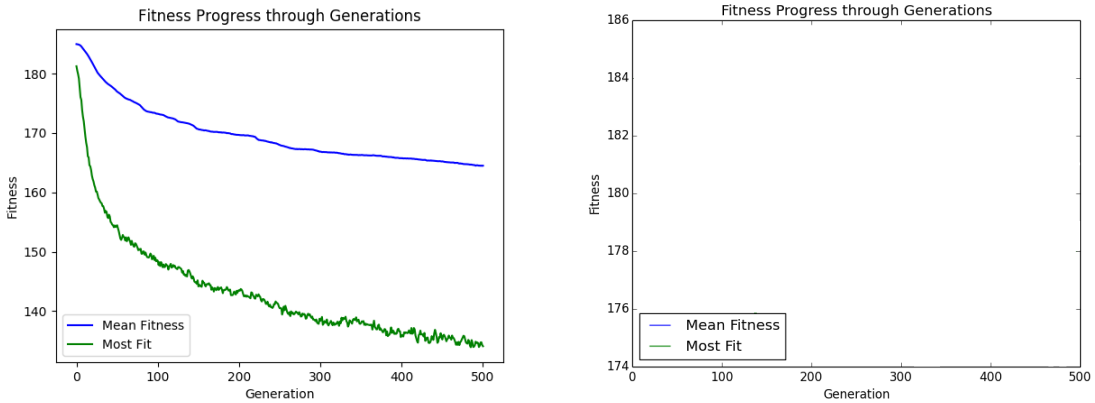
Figure 1: The 14x14 Game Board



(a) Mean Fitness across Generations

(b) Mean Final Score and Run Time

Figure 2: Varying Population Size



(a) Control

(b) Final Algorithm

Figure 3: Fitness of final algorithms.