Evolutionary Computation

Y1481702

January 24, 2018

Contents

Abstract

This paper develops a new evolutionary algorithm to create a controller for the classic computer game, Snake, using the DEAP Python framework. Implementation trade-offs and decisions are explained and justified using a mixture of experimental data and existing literature. The new implementation critiques and builds on existing work in both evolutionary computation and Snake AI players.

1 Introduction

1.1 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]

1.2 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.

1.3 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.

The task provides some sensing functions are given that tell the snake if there is a wall, itself or food directly ahead. Additional sensing functions may, and will, be implemented in order to bring out important features of the game state that will require action to be taken. The brief states that no additional movement functions can be implemented, which relates to the fact that the controls of the game are fixed. It should be noted that while the movement functions have not been altered, some of the gameplay code has been refactored to allow the new movement functions to take advantage of existing code.

2 Methods

2.1 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.

2.2 Representation

2.2.1 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. Having a solution that is too abstract may reduce some required detail from the problem and remove the ability to produce good solutions. The aim will be to encode the problem in a way that

the representation requires minimal storage space and can easily search the problem space for good individuals.

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. Deciding on important parameters and how an individual would react to different values of them seems to be a significant challenge of this type of implementation. It would tie the algorithm more closely to the game and reduce its ability to generalise.

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 re-runs the algorithm starting with the best individuals and a new population if no good results are found.

Neuroevolution
Neuroevolution is a hybrid technique between evolutionary computing and artificial neural networks (ANN). The weights and structure of the neural network are initialised and then developed using an evolutionary algorithm. Unfortunately as the brief requires that the DEAP Python library is used, neuroevolution is not an available option as DEAP does not yet support it. While DEAP's open data structures and transparency allow for virtually any algorithm to be implemented, it will be easier far to use the built in structures.

2.2.2 Choice and Justification

Genetic Programming will be used as it most closely fits the aims set out in the introduction above. It intuitively maps to the problem, and will be the easiest to implement as the brief already contains a starter function set, which may be expanded, and a terminal set. 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. A population of 1000 should be enough to evolve a reasonable Snake player and has a more reasonable runtime (Fig. 3b). Experiments have shown that a population of 1000 should converge near a player in around 75 generations (Fig.

3a). The data for a full run of the control has been provided for completeness to compare this implementation with the original paper (Fig. 2). 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. In general, the main measure used to compare individuals in this report will be their mean score on 500 games of the Snake once evolution is complete. This will give a fair representation of the real world performance of the individual. This measure is refered to as the 'Mean Test Run Score' and will also averaged over 30 code runs.

2.2.3 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 2.4.

Terminal Set
The terminal set uses the movement functions which, as previously mentioned, may not be altered. These functions change the directions of the snake to up, down, left and right. However, an additional terminal has been added which makes no changes to the current movement of the snake. This terminal should allow the snake to reduce its size by removing the need to check its current direction in order to maintain its current course.

Function Set
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.

While the brief limits new movement functions due to the way the game must be controlled by the player, taking no action is not strictly prohibited. In this case, both analysis of the program tree and practical experiments showed that by replacing the snake's ability to sense its direction with a terminal that allows it to maintain its current course resulted in a significant score boost without significant change in the program size (Table 2).

2.3 Population

2.3.1 Initialisation

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.

2.3.2 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]. A number of recent papers have shown improved results when using dynamic population sizes[8, 9], but these have focused particularly on genetic algorithms rather than genetic programming. Furthermore, DEAP does not have explicit support for dynamic population sizes and attempting to use them would significantly complicate the implementation without any guarantee of improved results.

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 3a. After each run, the most fit individual was used to play 500 independent game, these results are shown in Table 1 and by Figure 3b. 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 2.6.1. Another interesting feature in the data is the significant increase in standard deviation as the population size increases. This shows that while larger pop-

ulations 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[10]. A population of 1,000 seems to give a solution that is reasonably quick to converge (approx. 50 generations, Fig. 3a) but also produces a reasonable score across multiple games (around 6.97, Table 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.

2.3.3 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, covering more of this search space than one that is not. Ensuring a diverse population is important to allow the algorithm to explore several promising areas of this space at once[11]. Having a large enough population can contribute to having a good initial diversity, but diversity needs to also be maintained throughout the run of the algorithm to prevent premature convergence. Diversity is difficult to measure directly but a range of different measures including the number of different fitness values, number of different phenotypes/genotypes or entropy may be used as a proxy for it in different situations[5].

2.4 Fitness 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 2.3.2. In order to ensure a fairer representation of fitness, each solution will be evaluated across multiple games. It should be noted that evaluating an individual over multiple games and using the cumulative score achieves a significantly

greater average fitness, however it also significantly increases the runtime, which is not desirable.

Incentives
Certain scenarios in the game can lead to difficulties in evolving an individual that gets past a certain score. Rewarding individuals for making progress towards this, even if the score is not actually increased, can help to selectively breed programs that get past these boundaries. Likewise, punishing individuals that make obvious mistakes can also help to ensure they do not pass on these traits.

A number of incentives and penalties on individuals were considered:

- Evaluating the fitness with multiple games, as described above will incentivise the snake to perform well generally rather than in only one particular fluke case.
- Adding a significant penalty to the score for crashing into a wall will ensure individuals never take this choice.

2.5 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 finding a good solution in a reasonable time while preventing premature convergence.

Parent Selection has an element of randomness, in order to model external factors that occur in biological evolution. This randomness means that good solutions might die out in order to make fresh solutions. In order to keep track of the best solutions seen so far, DEAP's Hall of Fame has been used. This will allow us to retrieve the most fit solution that has ever been present in the population once the algorithm finishes running. As we are only evaluating the most fit individual for each run, the Hall of Fame size can be set to 1, but this can be adjusted as desired. The two most common forms of parent selection are roulette wheel and tournament. In DEAP, roulette wheel cannot be used when fitness values may be 0, or when the algorithm is minimising, this is a significant limitation, so tournament will be used for this implementation.

2.5.1 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[12]. The real aim here is to find the smallest possible program tree for a particular fitness.

All forms of bloat control have been shown to work best when augmented with a form of tree depth limiting[13, 14]. In this case, the program trees have been limited to a height of 8, a value decided using experimentation. 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. Traditional parsimony presure is parametric and therefore a model that specifies an exact trade-off between values of model and size is needed. This is difficult to produce as this value may not be known before runtime and may not be the same between different individuals[13].

Double tournaments can be used for parent selection as an extension of regular tournament selection described in Section 2.5. These are easier to parameterise as they do not require an exact known trade-off between program size and an individual's fitness. As such, double tournaments have been used to control bloat in this implementation. Experimentation was used to find reasonable values for the fitness tournament size and the size selection pressure. A static limit has also been used to limit the tree height to 7 nodes. Both of these values help to produce smaller program tress that have roughly equivalent fitness, as shown by Figure 4. The order of a double tournament has been shown to have no significant effect[13].

2.6 Variation Operators

2.6.1 Mutation

Mutation is the operator which incorporates new ideas into the population. Without it, genes would only be shuffled around and no novel solutions could be found. If no mutation exists, the success of evolution relies entirely on the initial population containing the required genes to produce a good solution. Too much mutation, however, will result in a highly unstable algorithm, so a trade-off is required here. Care needs to be taken when implementing mutation so as not to create biases. This is particularly the case with real-valued mutation. As this implementation uses GP with no real-valued terminals, this will not be an issue here.

2.6.2 Crossover

Crossover allows individuals to pass over parts of their genome to the future population. This is important so that two highly fit individuals can mix their genes with the aim of producing children which have an even higher fitness value. Out of the box, DEAP only supports one point-crossover for genetic programming. This should be suitable for this task.

2.6.3 Calibration

Extensive experimentation (Fig. 5) has shown that for this population size (1000), reasonably high values for both mutation and crossover tend to produce individuals that work best in the game. Particlularly good values appear to be produced when each value is 0.75, so this will be used. As discussed in Section 2.2.2, 75 generations seems to be enough for this population size to converge, so this will be our termination condition.

3 Results

While the control algorithm was originally intended to have a population of 10,000 and run for 500 generations, in order to make the following comparisons fair both algorithms have been run with a population of 1,000 across 75 generations. However, as can be seen from Tables 1 and 3, the final algorithm produces a better mean score with a population of 1,000 over 75 generations than the control did for a population of 10,000 over 250 generations.

Scoring
While Figure 6b shows no significant change in overall maximum fitness of the individuals at any point in the evolution, the new implementation is being averaged over multiple games and therefore it must perform well consistently to achieve this score. These individuals are fitter overall, and hence when they reproduce, they pass their abilities along to the next generation, as shown by the mean fitness of the population in Figure 6a. As such, the final algorithm proclaims a far greater final score in test games.

A Wilcoxon rank sum test has been used to compare the scoring distribution for the final implementation, S_F, with the original control, S_C. This test is used as there is insufficient evidence to prove whether or not either of the distributions S_F or S_C are normal.

The null hypothesis, H_0, states that the new algorithm produces results that are comparable to the control as they are pulled from the same distribution.

$$H_0 : S_F = S_C$$

Therefore, the alternative hypothesis is-

$$H_1 : S_F > S_C$$

We assume that H_0 is true and attempt to disprove this using the Mann-Whitney U test. The Mann-Whitney U test is significant if u is less than the critical value U, which is 317 for two sample sizes of 30, and if the probability that H_0 is true, $p(H_0) < 0.05$. The test returns a u value of 47 and a p value of 2.67×10^{-9}. Therefore there is sufficient evidence to reject null hypothesis.

4 Conclusion

4.1 Main Findings

This report has detailed the major trade-offs involved in evolving a controller for Snake, especially that of performance against runtime. The design decisions taken here have been justified within the context of the task- a short term university assessment for demonstrating knowledge of Evolutionary Computing. As part of this, having a reasonable runtime was a key requirement in order to perform all the necessary testing within the allocated time. Given this significant constraint, various parameters including mutation/crossover probability and population size have been calibrated in order to produce an algorithm that offers a reasonable performance. Statistics has been produced and included that will allow for easy re-calibration of the algorithm for their own requirements and compute availability.

DEAP has given a large amount of flexibility to the project and ensured that commonly used genetic programming features need not be re-implemented. However, it has provided some limitations. It does not have support for several aspects of Evolutionary Computing that have been discussed in this paper, such as neuroevolution and dynamic population size. These could be considered as future enhancements to the framework.

4.2 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.

Alternative approaches to this problem are a promising avenue for future exploration. For example, a snake that cycles through every square on the board 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.

5 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] A. E. Eiben, E. Marchiori, and V. Valko, "Evolutionary algorithms with on-the-fly population size adjustment," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2004, pp. 41–50.

[9] K. C. Tan, T. H. Lee, and E. F. Khor, "Evolutionary algorithms with dynamic population size and local exploration for multiobjective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 6, pp. 565–588, 2001.

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

[11] D. Gupta and S. Ghafir, "An overview of methods maintaining diversity in genetic algorithms," *International journal of emerging technology and advanced engineering*, vol. 2, no. 5, pp. 56 – 60, 2012.

[12] 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

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

[14] ——, "A comparison of bloat control methods for genetic programming," *Evolutionary Computation*, vol. 14, no. 3, pp. 309–344, 2006.

6 Appendix

Population Size	Run Time		Mean Test Run Score	
100	Mean	6.59	Mean	2.76
	Max	13.50	Max	11.60
	St.Dev	2.52	St.Dev	2.14
250	Mean	17.70	Mean	4.67
	Max	28.90	Max	18.8
	St.Dev	4.22	St.Dev	3.48
500	Mean	43.39	Mean	5.81
	Max	121.75	Max	21.97
	St.Dev	18.55	St.Dev	3.99
750	Mean	66.50	Mean	7.31
	Max	252.96	Max	23.90
	St.Dev	38.66	St.Dev	4.46
1000	Mean	78.59	Mean	6.97
	Max	133.16	Max	24.20
	St.Dev	21.63	St.Dev	4.61
2,500	Mean	249.24	Mean	10.73
	Max	518.06	Max	29.40
	St.Dev	83.79	St.Dev	5.73
5,000	Mean	572.78	Mean	13.99
	Max	997.97	Max	33.83
	St.Dev	185.05	St.Dev	6.64
10000	Mean	1437.17	Mean	18.0
	Max	39.27	Max	39.30
	St.Dev	7.59	St.Dev	7.59

Table 1: Population Size Performance over 250 Generations

Description	Mean Test Run Score		Program Size	
Control Function Set	Mean	6.58	Mean	44.80
	Max	24.00	Max	97.00
	St.Dev	4.61	St.Dev	16.75
Removed Direction Check, Added Maintain Course	Mean	13.97	Mean	48.73
	Max	33.23	Max	79.00
	St.Dev	6.72	St.Dev	12.71

Table 2: Comparison on different function sets

Description	Run Time		Mean Test Run Score	
Control	Mean	16.99	Mean	5.68
	Max	22.90	Max	45.00
	St.Dev	3.02	St.Dev	4.04
Final	Mean	89.48	Mean	14.45
	Max	198.20	Max	46.00
	St.Dev	39.97	St.Dev	6.51

Table 3: Performance of the Control and Final Algorithm

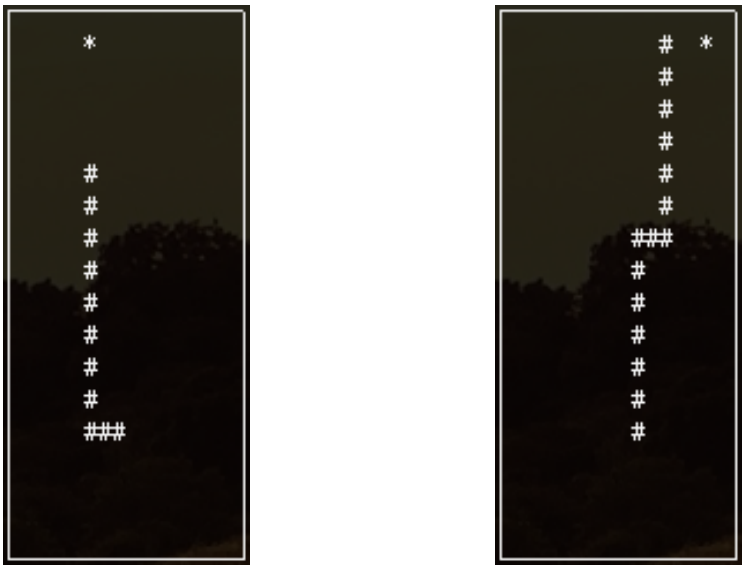


Figure 1: The 14x14 Game Board

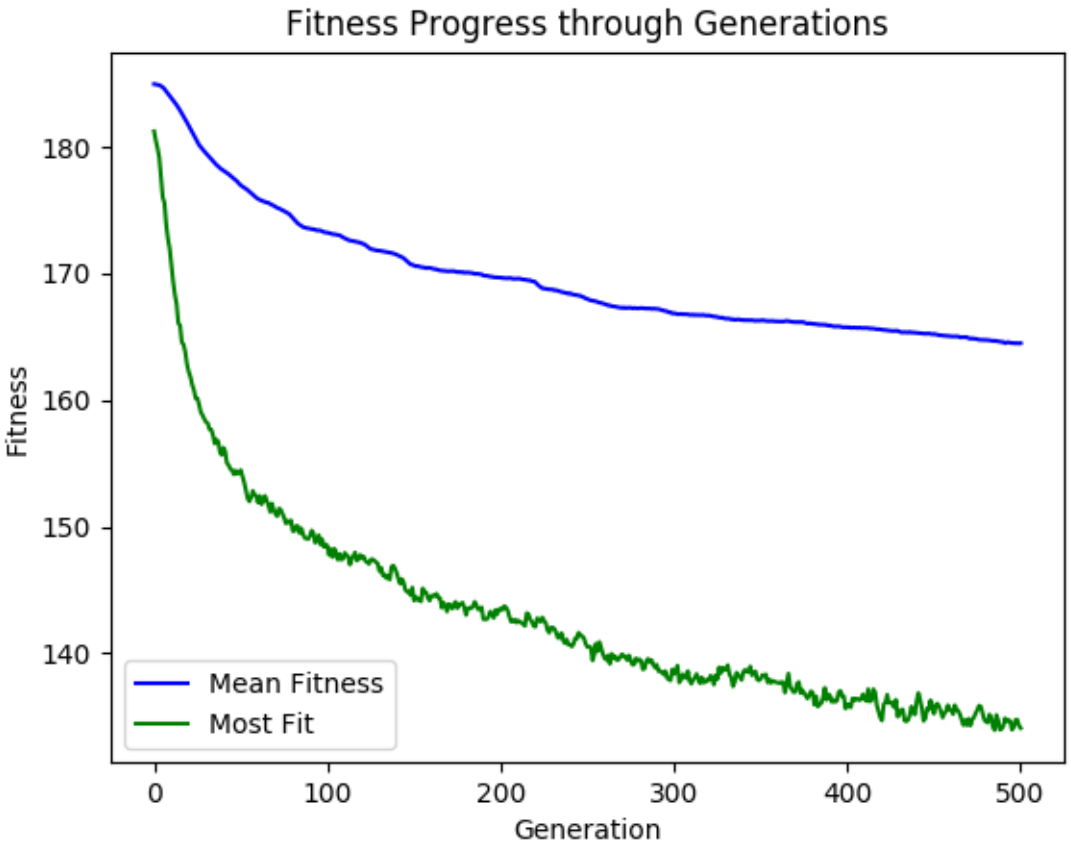
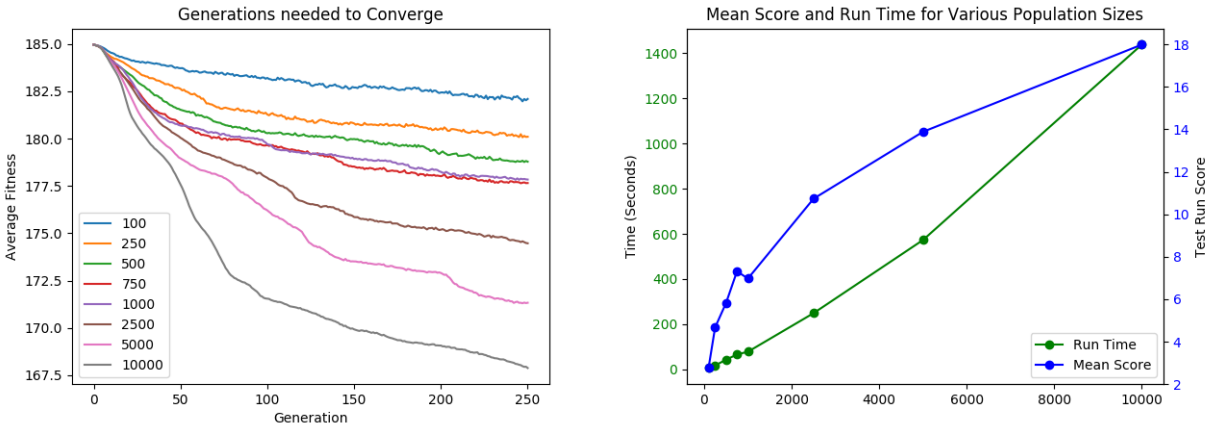


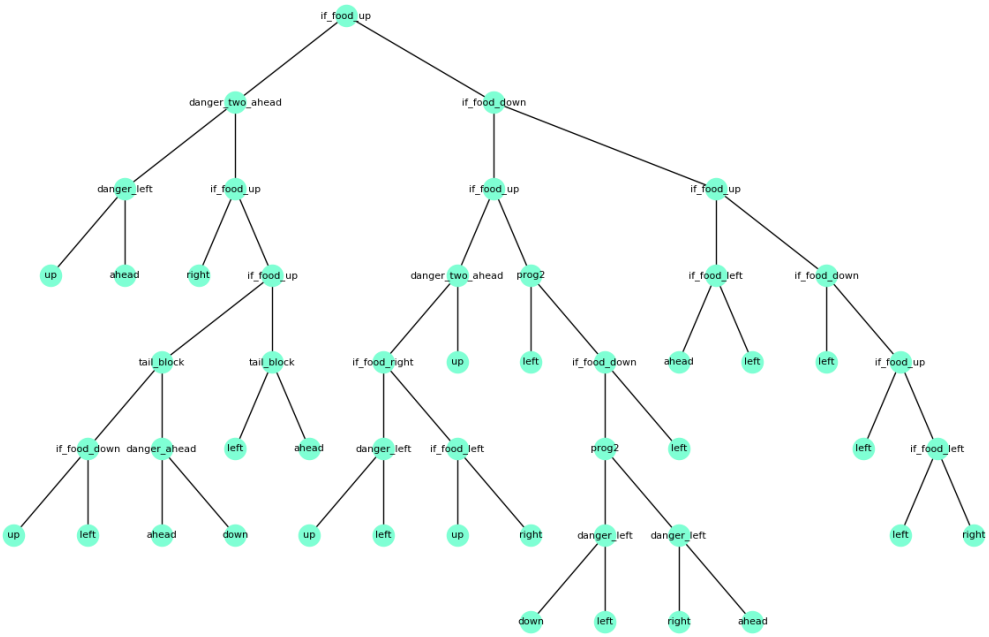
Figure 2: Control Evolution (10,000 Population)



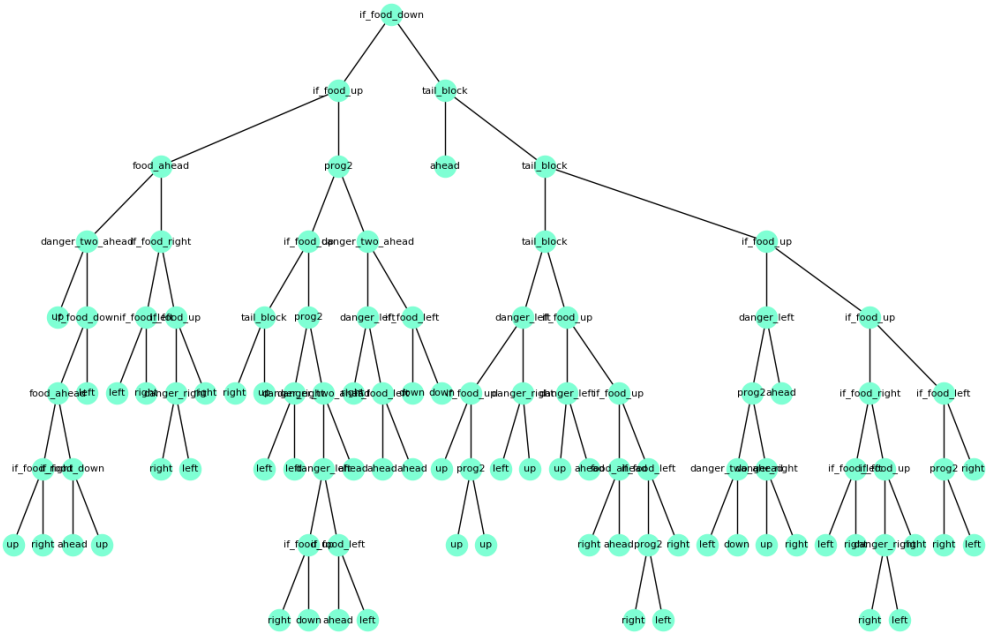
(a) Mean Fitness across Generations

(b) Mean Final Score and Run Time

Figure 3: Varying Population Size



(a) Double Tournament



(b) Height Limit (increased to 9) Only

Figure 4: How bloat affects Program Trees

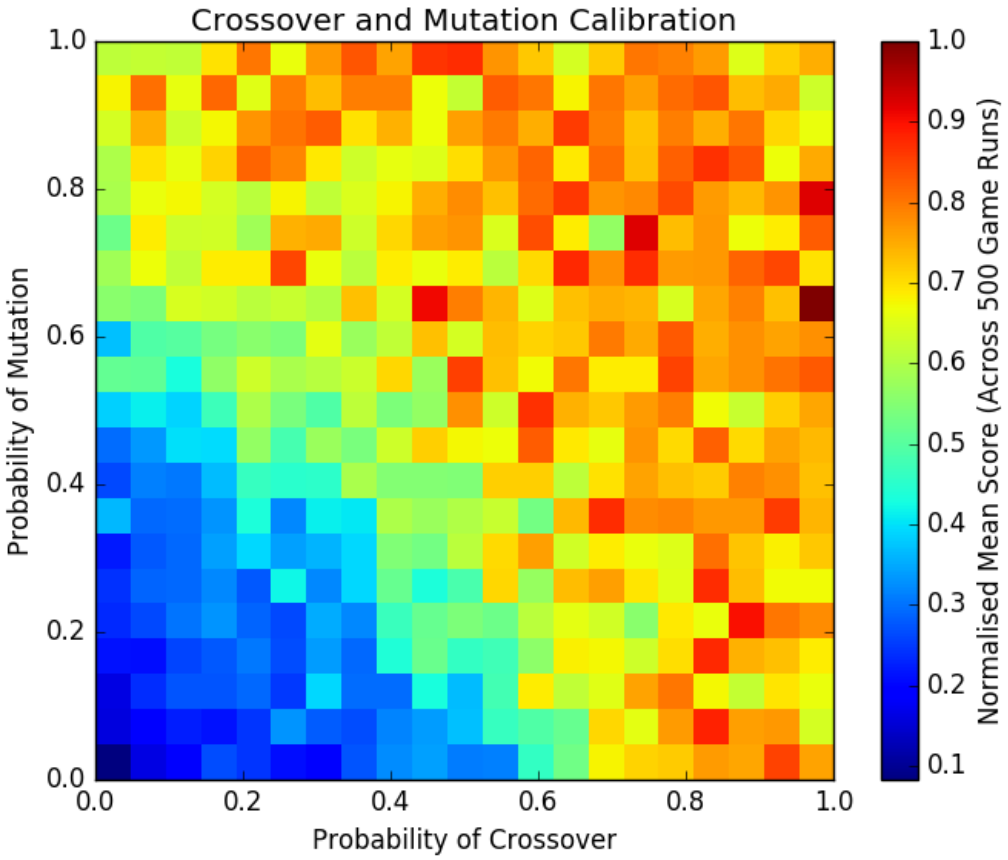
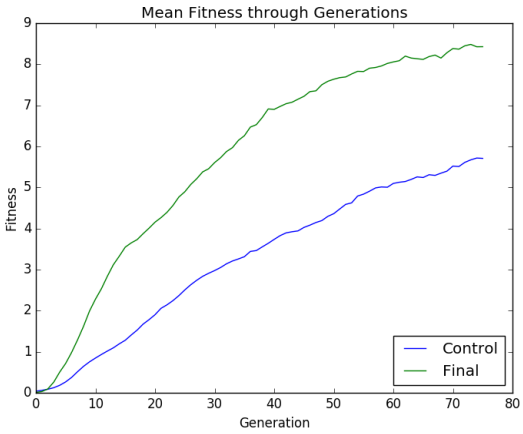
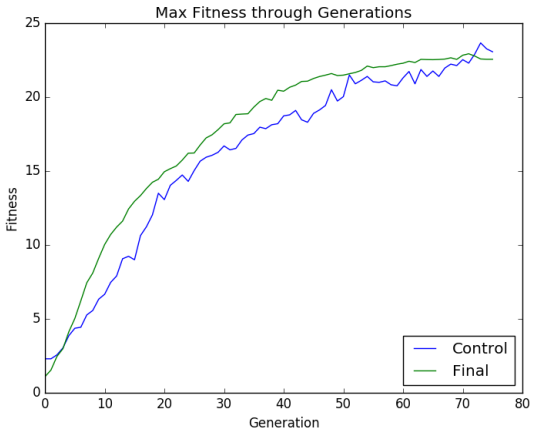


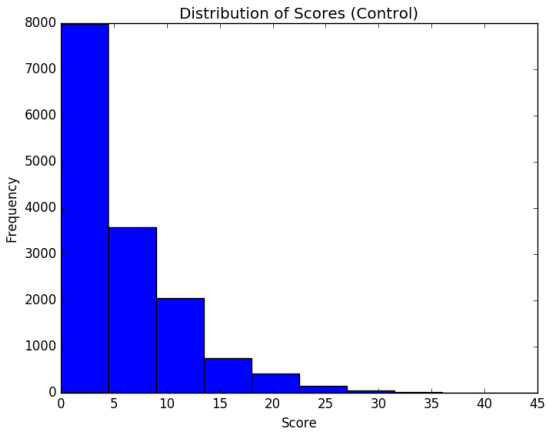
Figure 5: Varying Mutation and Crossover



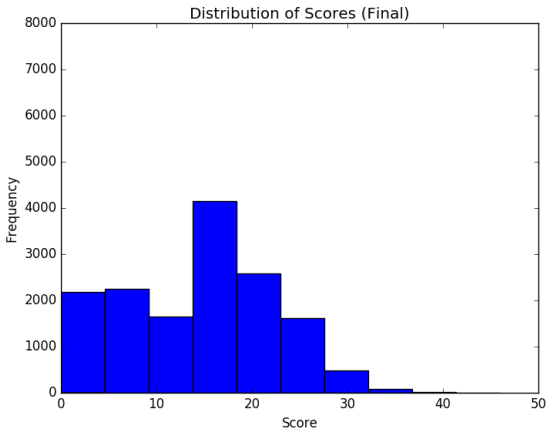
(a) Mean Fitness



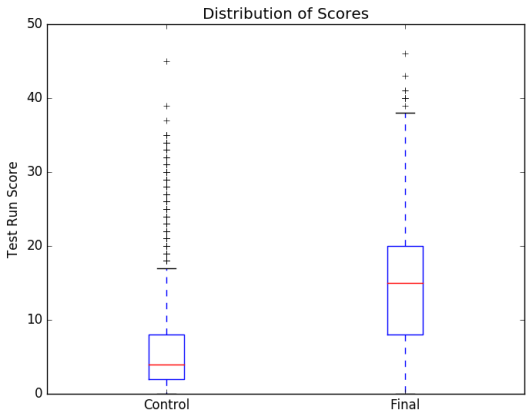
(b) Max Fitness



(c) Control Test Run Score Distribution



(d) Final Test Run Score Distribution



(e) Test Run Scores

Figure 6: Control and Final Algorithm