EVCO Open Assessment

Exam no: Y0076159

24th January 2017

Contents

List of Figures

List of Tables

1 Introduction & Problem Definition

The game of Snake was first invented in 1976, as an arcade game named 'Blockade', and later as a computer game named 'Worms' for the TRS-80 microcomputer in 1978 [12]; it gained a huge traction on mobile phones following Nokia engineer Taneli Armanto's implementation of the game for the Nokia N6110, released in 2002 [12]. The game, in which a single player controls a 'snake' composed of a dot, square or object, is named after the way in which the body of the dot, square or object trails behind the dot controlled by the player. The player can turn the snake's head to change it's direction of travel, whilst the forward movement is uniform relative to time (the player has no control over the speed at which the snake moves). The player aims to win the game by collecting food places on the screen, with every item of food eaten causing the snake's tail to grow. As the player eats more food, the tail grows longer, and it becomes more difficult to navigate the plane without colliding with a surrounding wall, or the player's tail: these form the 2 losing conditions, whilst a player who successfully grows his snake to cover the entire plane has beaten the game.

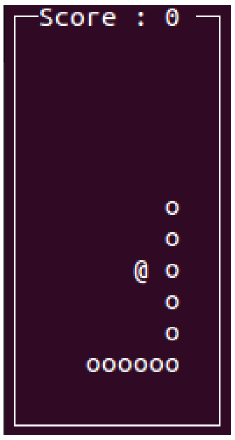

Figure 1: A game of snake as simulated in our environment.

Our version of the game is restricted to a 12x12 grid (144 spaces), surrounded by walls, with a single piece of food present on the board at any given time. The snakes starts with an initial length of 11, and is placed facing east with the head at coordinates [11, 4]. Furthermore, the snake can go in any one of 4 absolute directions (up, down, left, right), and always advances by 1 square at a time along either the X or Y axis of the board. As the snake is initialised with a body of size 11, the maximal score obtainable is $144 - 11 = 133$), at which point food cannot be placed on a tile not occupied by the snake's body.

Within the context of this project, we will be aiming to produce an evolutionary algorithm (i.e: an algorithm capable of producing a game-winning A.I. from a set of logical connectives and in-game actions) which can beat the game. That is, we will aim to maximise the number of pieces of food eaten in a run of the game. Key challenges will likely be:

- To ensure the snake does not collide into it's own tail or the surrounding walls.
- To ensure the snake successfully eats the next piece of food without timing out.
- To ensure the algorithm can be run relatively quickly, so it can be evaluated over a large number of runs for statistical analysis.

2 Literature review

i. Genetic algorithms

Evolutionary computing is a form of algorithms which seek to find a solution to a problem through a process similar to biological evolution. [2] The general concept involves initialising a set of candidate solutions, and iteratively updating them to obtain a 'fitter' solution. This is effectively implemented by assessing each candidate using a fitness function, before selecting a a subset of solutions which will be allowed to be bred into child solutions, mutated, and inserted into the next generation's population. This causes the overall fitness of the population to grow over the cycle of evolution, creating a candidate solution which will be closer to the global optimum. Algorithms which implement this evolutionary cycle are known as evolutionary algorithms. The field of evolutionary algorithms (EA) is sub-divided into a series of genetic representations and implementation systems. We will first discuss each of these, highlighting key papers and applications of each to our current situation.

Genetic algorithm

A genetic algorithm (GA's) traditionally represent the genotype of a candidate using a binary string, although other representations are possible [24]. It is extremely effective at tuning parameters of a model, as demonstrated by Wloch and Bentley [25] who used a GA to optimise the parameters of a simulated Formula One car, allowing it to minimise it's lap time. GA's have been implemented for use in the snake game by Yeh et al. [26]; the algorithm functioned by adjusting the rating functions of 4 operations, each of which decided the snake's next move based on either minimising the number of turns, the free space surrounding the head, or the presence of food near the snake's head. This enabled the snake to evolve a variety of solutions formed of varying frequencies of each function, resulting in a snake capable of beating the game (collecting the maximal number of food possible, where the snake's body fills the entire plane). As the snake's behaviour is not analysed by any means other than it's score in the game, it is difficult to know whether the solution reflected an EA's ability to create novel solutions to the problem. Due to this (and the restrictions on hard-coding solutions to this assessment), an approach using genetic programming was considered, as seen in section i..

Genetic programming

Genetic programming (GP) utilises a tree structure to define the genotype of a candidate solution [8]. Nodes in the tree represent certain operators, with leaf nodes representing operands. This allows a decision tree to be evolved, which can rely on conditionally verifying various functions, and applying a move depending on the values of these functions; an example of this is analysed in subsection iv.. Other examples of applications of GPs to games can be found in Hauptman and Sipper [13], who evolved a chess playing agent using a GP. It has also been used in other video games, such as a tower defence game [17], space invaders, froggers and missile command [15]. Within the context of defining a program, GP produce invalid states less frequently than genetic algorithms, as they can be strongly typed (such as in [15] to ensure that the syntax of the defined operations conforms to an executable standard [6]. As such, it provides an easily analysable solution to the creation of a snake A.I, as the output of a function is a readily human-readable output. It's additional efficiency in evolving further maximises it's capabilities towards our problem.

Grammatical Evolution

Grammatical evolution (GE) presents the next development in EA's; as with genetic programming, it aims to produce a section of code which can define a function, solving the problem at hand. As detailed by Ryan et al. [21], it aims to avoid the requirements of closure imposed on solutions in genetic programming, which are the main causes of bloating and limitations in crossovers. This is circumvented by the use of a user-specific grammar in Backus-Naur form, restricting the search space to only valid individuals. In contrast to a GP, it splits a genotype and phenotype; that is, the objects operated on by the search algorithm differ from those evaluted by the fitness function. This occurs as the grammar is converted into a program through the use of codons, which index the child element to select for each element in a BNF. As such, GE's provide an escape from the problems suffered by GP's, without adding any significant complexity: this is reflected by a significant improvement in performance when compared to GP's, as demonstrated by Michael O'Neill [19]. In this paper, GP's and GE's were implemented to solve the Santa Fe ant trail issue, demonstrating a greatly improved fitness for GE's when evolved over the same number of generations as GP's; this is visible in Figure 2. It is therefore apparent that a GE would be more appropriate than a GP to solve the Snake game, but as it is not natively available in the DEAP [9], implementing it would go beyond the scope of this project; therefore, a GP will be used.

Neuro-evolution

Neuro-evolution presents the third viable algorithm for the snake game. In this scheme, a neural network's topology and weights are evolved to develop a network which can sense various parameters from it's environment, and output an optimal action back into the game. This was well demonstrated by Hausknecht et al. [14], who evolved a single network capable of beating human high scores in a variety of Atari 2600 games. It has also been used to play simple board games, such as Tic-Tac-Toe [11], or GO [20]. As such neuro-evolution is an extremely potent methodology, and further work in it's implementation for the snake game is worthwhile, but it's unavailability in DEAP restricts it's usage for this project.

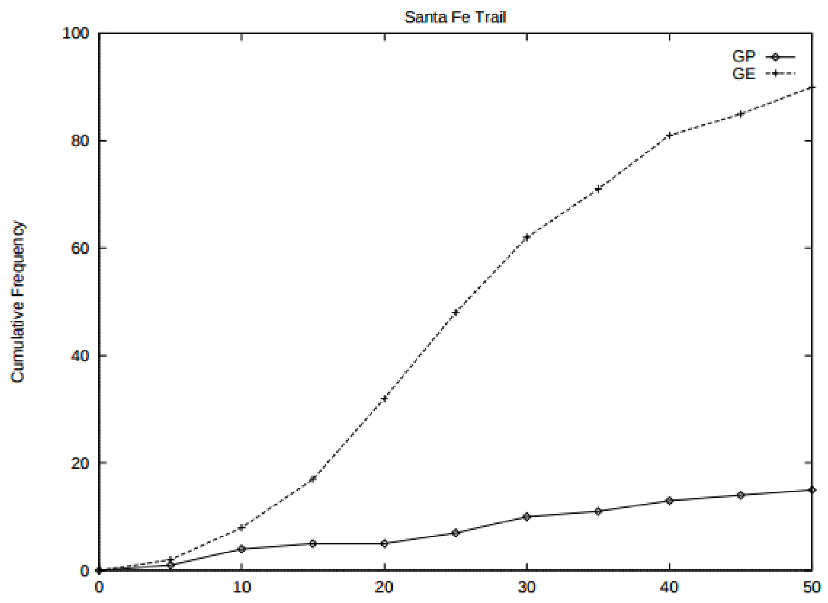

Figure 2: GE in comparison to GP, Source: [19]

Furthermore, we could hypothesise that a neuro-evolution approach to a simple problem such as the one at hand (relative to playing 61 different games with the same network) may result in a solution which is difficult to analyse, and potentially excessively large in comparison to a GP/GE approach.

ii. Co-evolution

Finally, the last applicable evolutionary technology considered for this application is co-evolution (differential evolution, gene expression programming and evolution strategies were also considered but found to be no more powerful than a GE or GP algorithm within the context of this problem). Co-evolution allows a second population to be evolved in parallel to the first, such that the fitness of an individual in one's population is evaluated using individuals from the other population. This can be combined with any of the previously discussed techniques by running a pair of the algorithms in tandem; however, as we will later detail, it is possible to develop a solution to the snake game which is independent from the size of the map, or the placement of food. As such, there is no competitive development which could be obtained from evolving a map, and therefore co-evolution would not be beneficial to this experiment.

iii. Applications of artificial intelligence to Snake

Some research has also been undertaken to develop intelligent agents for the snake game: Bowei Ma [4] has described an approach to reinforcement learning using Q-learning and SARSA through neural networks, but do not provide any applicable innovation to a GP-based implementation as they use a simple score-based fitness function and a completely different algorithm. Christopher Lockhart and Authors [7] applied back-propagation based learning on neural networks to the game of snake, and implements an improved fitness function which rewards the snake for every unit of time played in the game. Ehlis [10]'s work (analysed in subsection iv.) is most likely the most applicable research from which to develop our algorithm.

iv. Genetic programming for snake

Ehlis [10] has provided the only comprehensive article specific to the applications of genetic algorithms to Snake. The definitions of the problem diverge from our assessment in a few non-essential ways: snake movements can be inherently defined relative to the snake's current direction (i.e: a turn_right and turn_left terminal is defined), and the playing field of the game is a larger rectangle than the 1212 field. In all other respects, Ehlis provides an overview of a method which functions on the same game as our case. Given this, we will first analyse this approach to the problem, and aim to build an improvement both in terms of computational performance and quality of the solution.

A generic set of terminals allowing the snake to navigate the world is defined: Forward, Left and Right. The latter two allow the snake to make a change of direction relative to it's current heading, whilst the first argument does not affect the heading. The relative movement provided by the two turn operators does allow for a simpler evolution, as implementing an equivalent relative turn using absolute directions requires a significant number of operators, as we will later discuss subsection iv..

Non terminals can be categorised as either food sensing (ifFoodAhead, ifFoodUp, ifFoodRight) or danger sensing (ifDangerAhead, ifDangerRight, ifDangerLeft, ifDangerTwoAhead). The use of absolute food sensing (ifFoodUp, ifFoodRight) does however require the addition of 4 sensing methods to orient the snake, allowing it to undertake the correct action given the context. This introduces some complexity into the set of operators, potentially limiting the efficiency of the algorithm.

These sets of operators therefore define a generally greedy behaviour: the snake may be progressing towards the food, either at the root of the decision tree or further down once a given situation occurs (such as the 'wall-slitherer' describes in [10]). This evokes a classical approach to the game, similar to what we could expect from the average playerś strategy. However, given that the last move of a game-winning situation would involve fully covering the map, a greedy approach would need to be very attentive to surrounding dangers to correctly navigate a path to the wall. As such, the difficulty a greedier snake may have in adapting to every potential situation may be the cause for why the algorithm may not produce optimal solutions: an example situation is stated by Ehlis. This issue forms a basis for the design as later explained.

Some novel features are implemented in this algorithm: the use of multiple evaluations on every snake at every generation cause a larger cost in computational requirements, but do greatly improve the robustness of the solution to food placement and limit the effects of lucky/unlucky food placement on the performance of a snake. Similarly, the fitness function punishes snakes which do not find any pieces of food, thereby promoting snakes which search for the food. The algorithm also relies on priming the population of individuals, by selecting well-performing pre-produced solutions in the initial breeding pool.

Additional details of the algorithm are however lacking: a large population of 10000 individuals is evolved for 500 generations; the large population is potentially a counterweight to the lack of mutation, as unique genomes are created at the start of the evolution rather than during it; this most likely results in an equivalent effect. with a maximum tree size of 150 nodes to limit bloating. Crossover probability is stated as .10 for leaves, 0.80 for nodes, with a mutation probability of 0, in addition to the use of primed populations from previously unsuccessful runs. Additional details concerning which mutation or crossover functions were used are omitted, and reasonable assumptions of these are discussed in subsection v. to evaluate the method. The algorithm did however account for the randomisation of food placement, mitigating it by evaluating each individual over multiple runs per generation; this was also considered in our algorithm, as we will later discuss.

The proposed solution results in the creation of a game-winning decision tree (which scores the maximal 211 points). This algorithm was re-implemented to establish a benchmark for our research, revealing a significant requirement in computational time, as discussed in subsection v..

v. Implementation, assumptions and results

In order to obtain benchmark fitness values, tree sizes and computational times against which to evaluate our new algorithms, the algorithm developed by Ehlis was implemented using the Python DEAP [9] package; the code for this can be found in appendix C.

The algorithm was run 30 times in order to obtain a reliable statistic of it's performance when provided with the population size and computational resources proposed by Ehlis.

Given that approximately 30 hours of computational time were required to evaluate this, a subsequent evaluation (using a population of 1000, and 150 generations) was run to obtain results against which we can compare our new algorithms quickly. Some assumptions regarding the type of mutation were made: uniform tree insertion was first considered, but as the crossover was set to probability 0, this would be unable to grow the tree. Therefore, a uniform mutation which inserts a subtree of depth 1-3 was utilised. Crossover was set with a probability of 0.0 as per Ehlis's design. Finally, we assumed that the fitness tournament was used to drive the population's average fitness upwards: this was implemented with a size of 2, and that a standard evolution algorithm was used (DEAP's eaSimple).

When limited to a practical computational time (150 gens at 1000 population), the algorithm performs well, with an average of 97.97 pieces of food eaten, and a smaller standard deviation of 29.79 over 30 runs. This highlights the volatility of the algorithm, as well as the need for a large population or a larger number of generation to obtain an optimal value. One should note that the algorithm only has a 16.6% success rate in producing optimal (game beating) snakes; this is certainly related to the shortening of the algorithm by 350 generations, and the algorithm may therefore be more successful than the results above. This was demonstrated in [10], who finds that given a primed population it is possible to consistently an average max score of the population close to 200.

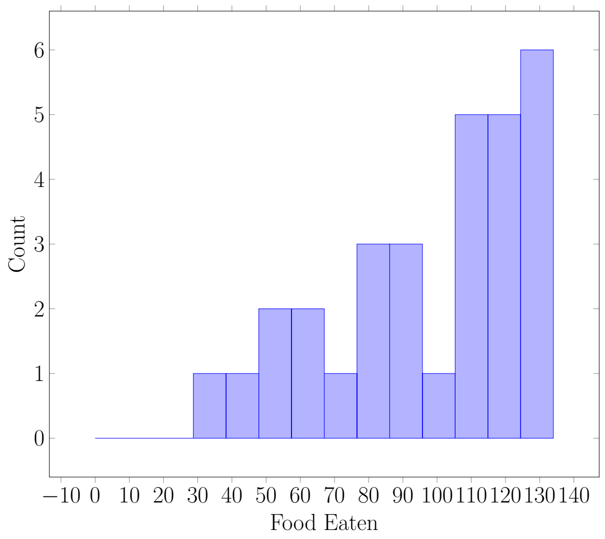

Figure 3: Histogram of best individuals from Ehlis's algorithm, 500 generations, 1000 population

This was verified using a second evaluation of the algorithm for 150 generations and 10 000 population, as seen in Figure 4. A double tournament was however used to ensure the evaluation could be undertaken given the available resources: this may be the cause of the larger standard deviation of the results, but we can note that as the median of the population is higher in the data with the larger population, we can expect the algorithm to perform. It is therefore possible that our final algorithm would perform better than the results detailed in this report if given a larger population or number of generations; this would have to be verified in the work's extension.

Nevertheless, we can establish a benchmark against which we can compare future algorithms, keeping in mind the scale of resources provided to the algorithms is limited given available computing power. This can most easily be described as an average best score of 97.97 (std: 29.79), given 150 generations with a population of 1000, with an average tree size of 362.8 nodes.

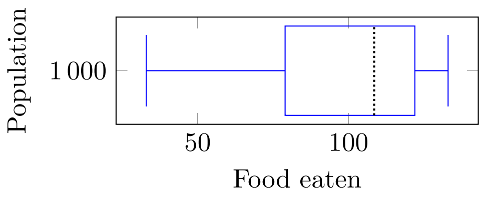

Figure 4: Results of Ehlis [10]'s algorithm over 30 runs.

3 Testing & Design methodology

In order to evaluate the effects of operators or fitness functions, each candidate fitness function will be evaluated 30 times, with a consistent configuration: a population of 1000 individuals evolved over 150 generations. These parameters allow us to evaluate the solutions rapidly, but results should be considered with the assumption that a larger population and number of generations would improve the score and consistency of the algorithm, in the same way our restricted implementation of Ehlis [10]'s algorithm underperforms compared to the extensive experiment performed in that report. We will aim to create an algorithm which requires less computational power, and produces optimal results more consistently with a smaller population and tree size. Fitness functions will be evaluated using the same algorithm (eaSimple), although a later comparison of algorithms will be run. Finally the optimal mutation and crossover probabilities will be found by iterating combinations of them with the most promising algorithm. As the benchmark algorithm discussed in subsection v. does not produce normally distributed data (as visually verifiable using the graph), a Mann-Whitney U-test will be used to compare the score of subsequent algorithms, as it does not rely on the assumption of normal distributions[23]. The two tailed variation of the U-test will be used, to ensure both algorithms can be found to be statistically better than the other. A p value of 0.05 will be used as a level of statistical significance.

4 Design & implementation

i. Evolutionary algorithm

A genetic programming algorithm was used, as it provided the most flexibility available from the DEAP package in terms of the solution space explored as it utilised a large number of operators, each with a small individual impact over the snake's movement, in comparison to an implementation which utilised a genetic algorithm such as Yeh et al. [26].

ii. GP Algorithm selection

Two categories of complete algorithms were considered for this experiment: DEAP's eaSimple, which implements the basic evolutionary algorithm as defined in Bäck et al. [3], and the newer lambda algorithms which implement forms of elitism (eaMuCommaLambda and eaMuPlusLambda). DEAP's eaSimple was chosen to increase the consistency of the algorithm, as it does not force elitism in the early generations of the algorithm; this permitted a larger variety of genotypes, thereby maintaining a diverse population of individuals throughout the evolution to maximise the probability of the correct mutations occuring. In contrast to this, we would expect the lambda-based algorithms to converge much more rapidly, although this effect can be managed using the parameters of the algorithm. Given the complexity of finding an additional 5 parameters of a lambda algorithm (mutation rate, crossover rate, number of individuals to select from current generation, number of children to produce at each generation), it would be most effective to implement a meta-genetic algorithm to find near-optimal parameters for a lambda algorithm. Given this data, and an equivalent optimisation of an eaSimple-based algorithm, we could conduct a statistical analysis of the two sets of data, and thereby find an optimal algorithm. However, for the purpose of this research, we will aim to optimise the parameters of an eaSimple and tournament selection so as to best cover the search space offered by both eaSimple and lambda algorithms.

iii. Strategy

As Ehlis [10] has already demonstrated a game-beating A.I, it was decided to focus on 3 aspects which we could compare to this algorithm:

- Equalling or beating the reliability of the algorithm
- Reducing the computational time
- Reducing the size of the average game-beating solution

Building on the work by Ehlis, we will aim to adapt his design to our requirements whilst improving it. As noted by Christopher Lockhart and Authors [7] in his publication about difference learning in Snake, "the greedy behaviour of [a learning agent] may be the cause of the agent's lack of high scores. Whilst this is disproven by the success rate of Ehlis's algorithm, we could argue that given a sufficiently large population, a set of operators can produce any solution which could also be produced by a subset of those operators. We attempted to demonstrate this in appendix A, where we compiled statistics from the best individuals of optimal (game-winning with a score of 133) and non-optimal solutions, across 30 runs with a population of 10 000 and 150 generations. As we can see, there is no noticeable difference between the frequency of operators, thereby demonstrating that not all optimal solutions are greedy, as this would be reflected by a lower percentage of food sensing functions (redundancies in the graphs (such as repeated operators or unrecheable branches) and noise in the data obtained may also have obscured any visible pattern). Nevertheless, we will hypothesise that developing a non-greedy snake may have the following benefits:

- A greedy snake which alters it's movement based on the positioning of food is likely to require additional sensing to avoid going causing game-ending situations, such as entering a dead-end to get closer to a piece of food. This would require the handling of many edge cases, which will cause the need for a large solution, which is therefore difficult to evolve. A non-greedy snake, which would aim to find a hamiltonian cycle through the map such that it's tail will never be in it's direct path until it has reached maximal size, could require a significantly smaller function to define it; this should make it easier to evolve, and therefore increase the reliability and speed of the algorithm.

- As a non-greedy snake would not sense food, it's fitness is unlikely to be greatly affected by the positioning of food. Therefore, it may not be necessary to evaluate each individual multiple times, which was done to improve the robustness of the solution in [10].

Due to this, it was decided to aim for a non-greedy solution which would not sense for food, but would rather aim to explore the map in a maximal way; this would adequately resolve the prescribed goal of beating the game by obtaining a maximal score, even though it may not be achieved in a minimalistic way.

iv. Terminal and Non-terminal set

The most significant difference of environment between our project and previous work involves the added complexity of using only absolute turning: as such, instead of a simple `turn_left` and `turn_right` function, we implement changeDirectionUp, changeDirectionRight, changeDirectionDown, changeDirectionLeft; these allow the snake's head to progress towards one of 4 directions at the next move. As these directions are difficult to use without any knowledge of the current heading (e.g: turning down when going north would cause the snake to reverse into it's own body), an additional 4 senses are used: `if_moving_up`, `if_moving_down`, `if_moving_left`, and `if_moving_right`.

As we will aim for a non-greedy solution, food sensing is not implemented in the new algorithm. We will aim to produce a hamiltonian cycle (where every space of the bordered plane is visited only once before returning to the original), as this will avoid having to take the snake's tail into consideration. Therefore, we will implement 4 wall-sensing functions: `sense_wall_ahead, sense_wall_left, sense_wall_right & sense_wall_2_away`. The first 3 of these are relative to the snake's orientation, and allow the snake to navigate against a wall, whereas the 4th sense allows the snake to avoid completely traversing the map along either the X or Y axis, thereby permitting it to avoid cutting itself off from part of the map which will be made available as it continues to move. This represents a generalisation of two operators proposed by Christopher Lockhart and Authors [7]: danger forward two, and danger right two.

Terminal	**Non-terminal**
go_straight	if_wall_2_away
go_up	if_wall_left
go_down	if_wall_right
go_right	if_wall_ahead
go_left	if_moving_up
	if_moving_down
	if_moving_left
	if_moving_right

Table 1: Operators

Some additional tests were conducted to determine alternative operators to `if_wall_2_away`) (set 3): more specific operators (such as `if_wall_2_ahead` or `if_wall_2_right`) were evaluated in place of the aforementioned operator, and resulted in a worse performance. This was evaluated over 30 runs with each set of operators, using the fitness function defined in subsection v., with a crossover probability of 0.8, a mutation probability of 0.7, a doubleTournament selection of size 5 with a parsimony value of 1.05 and fitnessFirst=True, using DEAP's eaSimple GP algorithm. Other operators included in the evaluation were all 5 terminals, all 4 orientation sensing, and all wall sensing of distance 1 (`if_wall_ahead`, `if_wall_right` and `if_wall_left`).

	Set 1	Set 2	Set 3
Alternative wall sensing operators	`2_left`, `2_right`, `2_ahead`	`2_right`, `2_ahead`	`2_away`
Avg score	26.198	27.626	47.2915
STD	43.874	45.570	57.791
Optimal runs	1	3	6
Ratio of perfect scores	0.05	0.15	0.3

Table 2: Comparison of operator sets

This resulted in Table 2: as we can clearly see, the smaller, more generic operator set consisting of the aforementioned operators in addition to if_wall_2_away (which returns true if the snake's head is 2 spaces away from a wall in any direction) results in a consistently better fitness. This is reflected by the difference in averages, and is confirmed to be statistically significant by the Mann-Whitney U-test values: set 3 has a higher median than set 1, with a Z-score=2.489, U=180.5 and p=01278, and set 3 also has a higher median than set 2 with a Z-score=2.682, U=170.5, p=0.00736. The third set is therefore statistically superior to other proposed operator sets, and will be used to evaluate in our algorithm, as detailed in Table 1. These were not implemented as a strongly typed primitive set, as there were no variations in the types of terminal or non-terminal operators which could cause un-evaluable solutions. Additionally, the conditions were implemented as part of an if_statement structure as their use in any other part of the tree would result in an invalid individual, unless a strongly typed set was implemented (which would have resulted in an additional overhead).

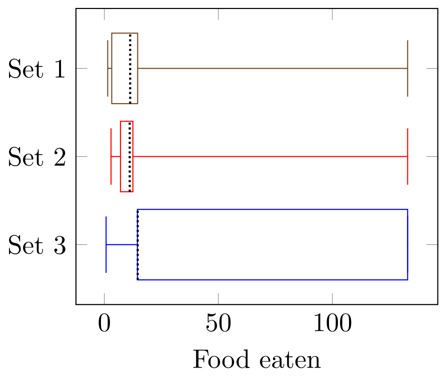

Figure 5: Comparative plots of operator sets described in Table 2

v. Fitness function

Keeping in line with the aim of a greedy snake, our fitness function does not include the snake's score in the game; this is due to the fact that snakes cannot sense food, and therefore their score will be directly correlated with the distance travelled by the snake (and therefore it's distance travelled, as it moves at a constant speed). In addition, as the snake's score can greatly be influenced by the random placement of food in the starting few generations, removing the scoring of food reduces the skewing of a snake's fitness due to lucky food placing. This further reduces the need to evaluate each snake multiple times, as the snake only reacts relative to constant objects in the plane (4 walls). Finally, to ensure the hamiltonian cycle covers every square of the map, we introduce the notion of a cover to the fitness function, which is equal to the proportion of the map visited by the snake during the run. This ensures that snakes develop a maximal coverage of the map, so that it can grow to a maximal size.

We can therefore define the fitness function as $fitness = (\%\ of\ map\ covered) + (steps/100)$.
In addition to this, we also implemented a penalty for snakes which enter a non-maximal loop: this is implemented by checking if the snake has timed out (i.e has travelled 196 steps without collecting food). This removes problematic solutions from the early stages of evolution, as these individuals would otherwise score highly due to their long survival; instead, they will now be assigned a fitness of -10, thereby greatly reducing their likelihood of selection, and therefore their effect on the population's genotype.

vi. Creation of individuals

vii. Selection, Mutation & Crossover

In contrast to Ehlis's algorithm, which relied on the insertion of tree structures in the place of individual nodes (using mutUniform) to grow a solution beyond it's initial size (as crossover probability was set to 0), our genetic algorithm utilises crossover throughout it's evolution. Therefore, it was possible to rely on crossover to grow a tree's size, whilst DEAP's mutNodeReplacement operator, which replaces a primitive from an individual by any other primitive which has the same number of arguments, was used to mutate individuals. This allows nodes within the tree to be replaced with more appropriate nodes, thereby improving their fitness. The basic one point crossover operator was used, without bias for leaf nodes.

As details of the selection algorithm in [10] are sparse, we decided to utilise a double tournament, to ensure that the size of the solutions was minimised given an equal fitness. As such, DEAP's double-Tournament implementation allows the size of individuals to be minimised without greatly hampering the fitness of the population. The parameters for the mutation rate, crossover rate, tournament size and parsimony can be found in subsection viii..

viii. Parameter selection

In order to obtain a tuple of parameters (crossover probability, mutation probability, selection tournament size, parsimony), a search across pairs of crossover and mutation probabilites was undertaken. This was implemented using the set of operators defined in Table 1, using DEAP's eaSimple algorithm Each combination of probabilities (in increments of 10%) was evaluated 5 times, and it's average was plotted in Figure 12. Additional runs would have been beneficial to the overal smoothness of the graph, as outlying runs may have a strong impact on the averaged value. Nevertheless, an optimal mutation rate of 0.8 and a crossover rate of 0.7 were chosen as the best parameters. The same search was repeated over tournament size (range from 2 to 7 in integer increments) and parsimony (range from 1 to 2 in 0.05 increments), as seen in the 3d plot Figure 11. This yielded a best parsimony and selection tournament size of (1.05, 6), with an average 116.12 pieces of food eaten. The small parsimony size would certainly have allowed the individuals to maximise fitness rather than size, causing more bloat and slowing the solution, but increasing the number of optimal individuals in the run. The high fitness tournament size is effectively implementing a form of elitism, and we could expect that as we increase the tournament size we would approach the behaviour of the Mu-Comma-Lambda or Mu-Plus-Lambda algorithms (where the fittest n % of a population are maintained and reproduced to create a proportion of the next generation).

The population size was kept at 150 to allow for a statistical comparison of our algorithm within the timeframe of this project.

We should note that this method suffers from the assumption of independence between the two searches; that is, crossover probability and mutation rates affect the population's fitness independently to the selection tournament's fitness size and parsimony. This approach was selected to maximise the success of the algorithm given the constraints of this project; an alternative approach could be to implement a genetic algorithm which could alter these four parameters, and evolve an optimal parametrisation of our current algorithm. This method is known as a meta-genetic algorithm, and was succesfully used by Brain and Addicoat [5] to parametrise the crossover rate, mutation rate, tournament selection method and population size of another genetic algorithm. Therefore, the full set of parameters for our genetic algorithm could be determined by a meta-genetic algorithm, given sufficient computational power and time.

ix. Creation of individuals

Individuals were generated using the ramped half and half method proposed by Koza [16]. This algorithm randomly picks a value within the size range provided, and with .5 probability uses a GROW algorithm to generate a tree, or generates a full tree of that depth. [18]. While Luke and Panait [18] does not find any benefit of this algorithm towards the final fitness of the algorithm, the ability to avoid specifying an exact depth helps to reduce the dimensionality of the parameters of the algorithm; Luke and Panait does note that having a distribution of initial sizes increases the probability of generating good initial individuals. As such, it was decided to use the ramped half and half generation to produce a diverse set of individuals which we can then evolve into a solution.

x. Bloat control

Bloat control was implemented using decorators on the crossover and mutation functions, limiting the height of an individual's tree to 7. In addition, the use of a double tournament effected minimal pressure on the population, to avoid greatly hindering the algorithm's ability to find optimal solutions to the game. The combinations of these contributed to creating an algorithm which is quicker to run than previous illustrated [10]; this will be evaluated in subsection iii..

xi. Code optimisation

A series of optimisations were undertaken to improve the performance of our algorithms. Firstly, the timeout limit of a snake's run, which causes a snake to 'die' after a certain time has passed since it has last eaten food, was reduced to the size of the board. This allows earlier generations, where looping snakes occur the most, to end before any un-necessary computation occurs.
The body of the snake was re-implemented using python's Deque collection [1], which is optimised for fast

appends and pops on either end of the list-like container. The food positioning algorithm was improved, such that instead of randomly selecting a tile in the plane before checking it would be a valid space for food, the algorithm instead creates a list of free spaces, and randomly selects one from that list. This flattens the response time of the algorithm, which would most likely have been faster at the start of the game, but would require a significant number of queries before randomly finding an available tile. Finally, python's MultiProcessing module was added to the project, which allowed the evaluation of a generation of snakes to occur in parallel over a large number of cores, thereby greatly reducing the computational time required per generation. Multiprocessing was conditionally removed from the code to evaluate it's computational time relative to Ehlis [10], and is also un-used when the algorithm is provided with a seed (as this causes the execution to become non-deterministic).

xii. Algorithm summary

Our algorithm can be summarised as follows:

- Population: 1000, Generations: 150
- Mutation probability: 0.8, Crossover probability: 0.7
- Selection process: Double tournament, fitness first with tournament size of 6 and a parsimony of 1.05
- Crossover algorithm: cxOnePoint
- Mutation algorithm: mutNodeReplacement
- Maximum tree depth decorators: max depth of 7 on crossover and mutation.
- Creation of individuals: ramped (half and half) in range [2,6]
- Operator set: as described in Table 1.
- Fitness function: % of map covered + (number of steps taken/100), or -10 if the snake has timed out

5 Results and analysis

i. Executing the algorithm

The Python 2.7.6 code for our new algorithm can be found in `new_algo.py`. An optimal run can be obtained by running the following command: `python new_algo.py --seed 0.507299291138 --max_gen 50`

This will also display the optimal snake in game. Additional parameters can be passed to the algorithm (if no seed is specified):

- to run the algorithm multiple times (add arguments `--iterations n`),
- to run it for a different number of generations (`--num_gen n`),
- to run it using Python's multiprocessing module (`--multicore`, note that this results in a non-deterministic execution due to process-specific calls to Python's random module),
- to create a .csv and save the result of the execution (`--save_results`).

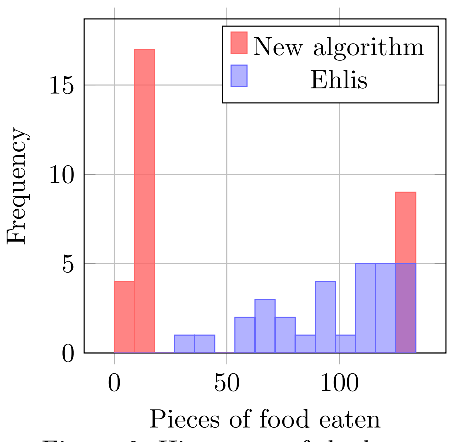

Figure 6: Histogram of the best individual in the population

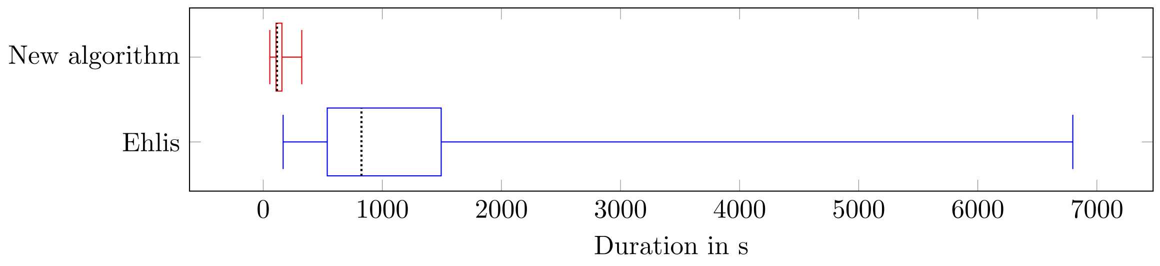

Figure 8: Average time required to execute the algorithm (150gens, 1000 population)

ii. Game score maximisation

Following the methodology defined in section 3, our new algorithm was evaluated 30 times using the best parameters found in subsection viii., and the results of this experiment were compared to our implementation of Ehlis [10]'s algorithm from subsection iv.. As we can see in Figure 7, our algorithm performs worse than our benchmark on average. After testing the distribution of both data sets for normality using a Shapiro-Wilk normality test [22] (benchmark data is not normally distributed with W=0.9127, p¡0.05, nor is our data with W=0.6211, p¡0.01), we confirmed the statistical significance between the algorithms using a Mann-Whitney U-test on the sets of best scores from runs of each algorithm (U=247.5, p-value of 0.00174) and the difference in median scores seen in Table 3. Our algorithm is therefore no better than our benchmark implementation in terms of the average max fitness. However, our new algorithm does produce a larger number of optimal solutions, at a rate of 0.34 vs the previous best of 0.167: this is reflected in Figure 6, where our algorithm forms two spikes at the bottom and top of the x axis.

The implementation of our non-greedy strategy through our fitness function and non-greedy operator set has resulted in a binary result: either a full hamiltonian cycle is found, which allows the snake to win the game regardless of the placement of food, or the snake fails to form a complete path and zig-zags the plane without leaving a path for it to travel back through the map. As such, we have demonstrated that a non-greedy solution can be as effective as a greedy approach to solving the Snake game, but may require a larger population or number of generations to achieve a consistent average score.

iii. Computational time reduction

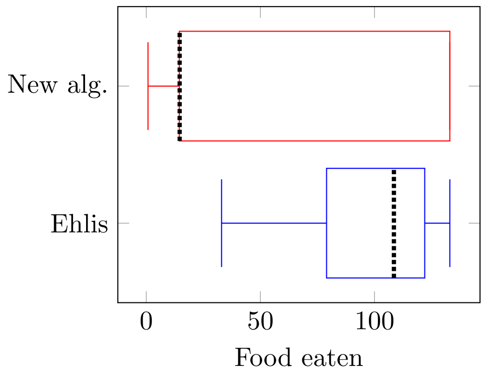

Figure 7: Comparison of algorithm performance

Our second objective involved reducing the computational time required to run the algorithm whilst maintaining a similar level of optimality. Whilst the latter objective was not achieved, our algorithm is significantly faster to execute, as detailed by the box plot in Figure 8. The median values were 824.9 seconds for the benchmark implementation, and 115.8 seconds for our algorithm. As neither data set is normally distributed (benchmark dataset: W=0.648 p¡0.01, our dataset: W=0.860, p¡0.01, using a Shapiro-Wilk normality test), a Mann-Whitney U test demonstrated that our algorithm is significantly faster with U=5 p¡0.0001. Therefore we may conclude that we have produced a significantly quicker algorithm, to the point where we may consider the following: given an equal amount of time, multiple executions (approximately 7) of our algorithm could be performed in the time taken by a single execution of the benchmark. Given the 0.33 percent chance of finding an optimal solution in a given run, we could expect every third execution of our algorithm to produce such results: we could therefore perform executions of our algorithm until finding an optimal solution, and we would expect this to take 3 runs, which is equivalent to half the computation time for a single run of the benchmark algorithm.

	Ehlis	New algorithm
Median	108.5	47.15
std	29.79294447	55.98871726
Proportion of optimal solutions	0.167	0.34

Table 3: Average best score (food eaten) over 30 executions

iv. Solution size minimisation

Finally, we were also able to obtain two datasets of the average sizes of individuals in the final population of each algorithm. These were found to not be normally distributed using a Shapiro-Wilk normality test (benchmark: W=0.926, p¡0.05, our dataset: W=0.924, p¡0.05), and a U-test confirmed that the difference in medians (91 nodes for our algorithm, 139.5 for the benchmark algorithm) is statistically significant (U=68, p¡0.01).

6 Summary & Conclusion

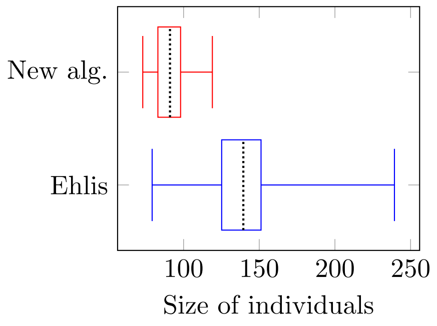

Figure 9: Average number of nodes in the final generation's population

In the course of this report, we have evaluated and implemented a genetic programming algorithm proposed by Ehlis [10] which aims to find a score-maximising solution to the game of Snake. This approach, which relied on a large computational power and high population size to achieve a reliable result, was discussed in subsection iv. and analysed in subsection v.; it was found to produce reliably good high-scoring snakes, in addition to a fraction of game-beating snakes, but required extremely large populations to function, and relied on mutation rather than crossover to evolve the solution.

Building on this, we have developed a minimalist, non-greedy subset of operators with the aim of creating a maximal hamiltonian cycle through the map. After finding a maximally optimal set of parameters given the resources available to this project, our algorithm was evaluated in terms of the fittest individuals it could produce, the computational time required to run the algorithm, and the average size of the final population. Compared to our benchmark, our algorithm was unable to produce an equally high consistant median fitness, but could produce optimal solutions (game-beating) more consistently. It was found that, given a restricted population size and number of generations, our algorithm resulted in a near-binary best individual which would either beat the game or consistently fail on the score of 14, due to the snake boxing itself into a corner without leaving any exit path. This was caused by the strategy undertaken throughout the project, as a greedy algorithm can incrementally learn to handle each situation correctly, whereas our algorithm would either learn a complete cycle, or fail to develop the correct mutation. This could potentially be remedied by using a higher mutation rate at later stages of the evolution.

Our algorithm does however produce more optimal solutions per run and per compoutational time, in addition to producing smaller solutions. These results would most likely be improved when run for a larger number of generations and a higher population, as this would increase the chance of finding a valid hamiltonian cycle throughout the game.

i. Further work

- Taking a population of optimal solutions, giving them more operators and evolving a more efficient A.I. which wins the game in the smallest time possible
- Testing the ability to improve the reliability of the algorithm using larger populations,
- Implement a meta-genetic algorithm to derive the set of operators, parameters, type of algorithm to use

Bibliography

[1] "8.3. collections — container datatypes — python 3.6.0 documentation," https://docs.python.org/3/library/collections.html, accessed: 2017-1-23.

[2] D. Ashlock, "Evolutionary computation for modeling and optimization by ashlock, daniel: Springer 9780387221960 hardcover - motor city books," https://www.abebooks.com/servlet/BookDetailsPL?bi=19887295686&searchurl=isbn%3D0387221964%26sortby%3D17, 2006, accessed: 2017-1-20.

[3] Bäck, , T. Fogel, , and D. B. Michalewicz, "Evolutionary computation 1 (basic algorithms and operators)," 1 Jan. 2000.

[4] M. T. Bowei Ma, "Exploration of reinforcement learning to SNAKE."

[5] Z. E. Brain and M. A. Addicoat, "Optimization of a genetic algorithm for searching molecular conformer space," *J. Chem. Phys.*, vol. 135, no. 17, p. 174106, 7 Nov. 2011.

[6] S.-H. Chen, *Genetic Algorithms and Genetic Programming in Computational Finance.* Springer Science & Business Media, 6 Dec. 2012.

[7] U. o. L. Christopher Lockhart and Authors, "Application of temporal difference learning to the game of snake," Ph.D. dissertation, University of Louisville, 2010.

[8] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *PROCEEDINGS OF AN INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS AND THEIR APPLICATIONS*, J. J. Grefenstette, Ed. U.S. Navy Center for Applied Research in Artificial Intelligence, pp. 183–187.

[9] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné, "Deap: A python framework for evolutionary algorithms," in *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '12. New York, NY, USA: ACM, 2012, pp. 85–92. [Online]. Available: http://doi.acm.org/10.1145/2330784.2330799

[10] T. Ehlis, "Application of genetic programming to the snake game - artificial intelligence - articles - articles - GameDev.net," https://www.gamedev.net/resources/_/technical/artificial-intelligence/application-of-genetic-programming-to-the-snake-r1175, 9 Aug. 2000, accessed: 2017-1-16.

[11] D. B. Fogel, "Using evolutionary programing to create neural networks that are capable of playing tic-tac-toe," in *IEEE International Conference on Neural Networks*, 1993, pp. 875–880 vol.2.

[12] G. Goggin, *Global Mobile Media.* Taylor & Francis, 14 Oct. 2010.

[13] A. Hauptman and M. Sipper, "GP-EndChess: Using genetic programming to evolve chess endgame players," in *Genetic Programming*, ser. Lecture Notes in Computer Science, M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, Eds. Springer Berlin Heidelberg, 30 Mar. 2005, pp. 120–131.

[14] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," *IEEE Trans. Comput. Intell. AI Games*, vol. 6, no. 4, pp. 355–366, Dec. 2014.

[15] B. Jia and M. Ebner, "A strongly typed GP-based video game player," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, Aug. 2015, pp. 299–305.

[16] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA, USA: MIT Press, 1992.

[17] L. C. Leong, G. K. Soon, T. T. Guan, C. K. On, R. Alfred, and P. Anthony, "Self-synthesized controllers for tower defense game using genetic programming," in *2013 IEEE International Conference on Control System, Computing and Engineering*, Nov. 2013, pp. 487–492.

[18] S. Luke and L. Panait, "A survey and comparison of tree generation algorithms," 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. 81–88.

[19] C. R. Michael O'Neill, "Evolving multi-line compilable C programs," in *In Proceedings of the Second European Workshop on Genetic Programming*, 1999.

[20] N. Richards, D. E. Moriarty, and R. Miikkulainen, "Evolving neural networks to play go," *Applied Intelligence*, vol. 8, no. 1, pp. 85–96, 1 Jan. 1998.

[21] C. Ryan, J. J. Collins, and M. O'Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Proceedings of the First European Workshop on Genetic Programming*, ser. LNCS, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, Eds., vol. 1391. Paris: Springer-Verlag, 14-15 Apr. 1998, pp. 83–96. [Online]. Available: http://www.lania.mx/~ccoello/eurogp98.ps.gz

[22] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[23] N. H. Timm, Ed., *Applied Multivariate Analysis:*, ser. Springer Texts in Statistics. Springer New York, 2002.

[24] D. Whitley, "A genetic algorithm tutorial," *Stat. Comput.*, vol. 4, no. 2, pp. 65–85, 1 Jun. 1994.

[25] K. Wloch and P. J. Bentley, "Optimising the performance of a formula one car using a genetic algorithm," in *Parallel Problem Solving from Nature - PPSN VIII*, ser. Lecture Notes in Computer Science, X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tiňo, A. Kabán, and H.-P. Schwefel, Eds. Springer Berlin Heidelberg, 18 Sep. 2004, pp. 702–711.

[26] J. F. Yeh, P. H. Su, S. H. Huang, and T. C. Chiang, "Snake game AI: Movement rating functions and evolutionary algorithm-based optimization," *researchgate.net*, 11 Jan. 2016.

A Difference in operator statistics between optimal and non-optimal solutions

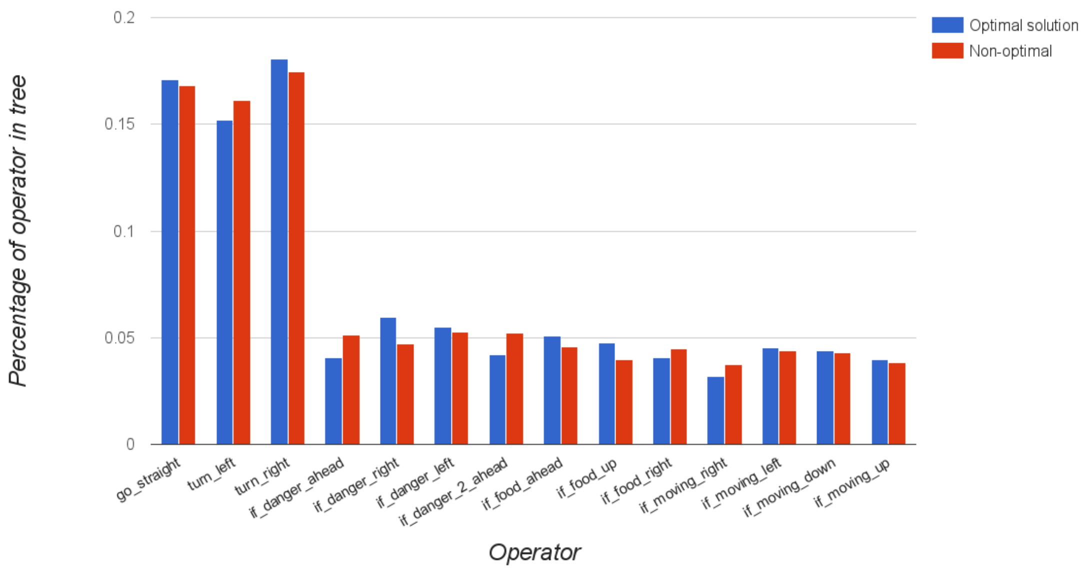

Figure 10: Difference in statistics of operators between optimal and non-optimal solutions

B Parameter search graphs

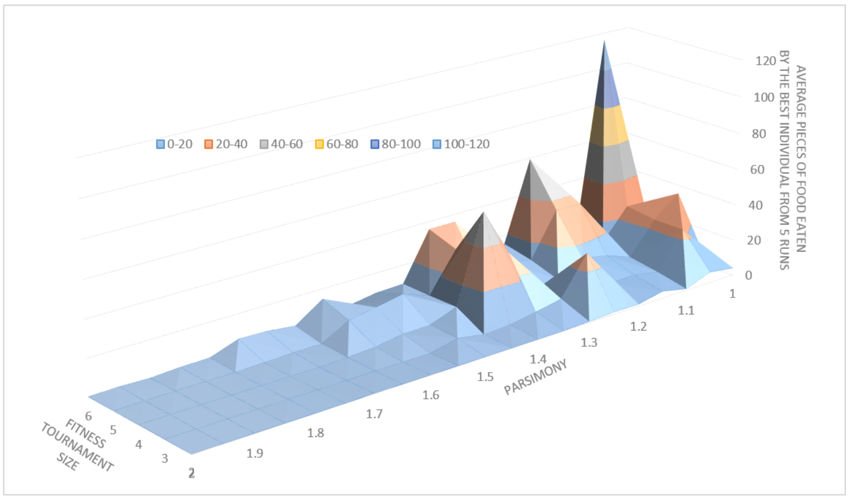

Figure 11: Average of 5 runs over the space of selection tournament sizes and parsimony values

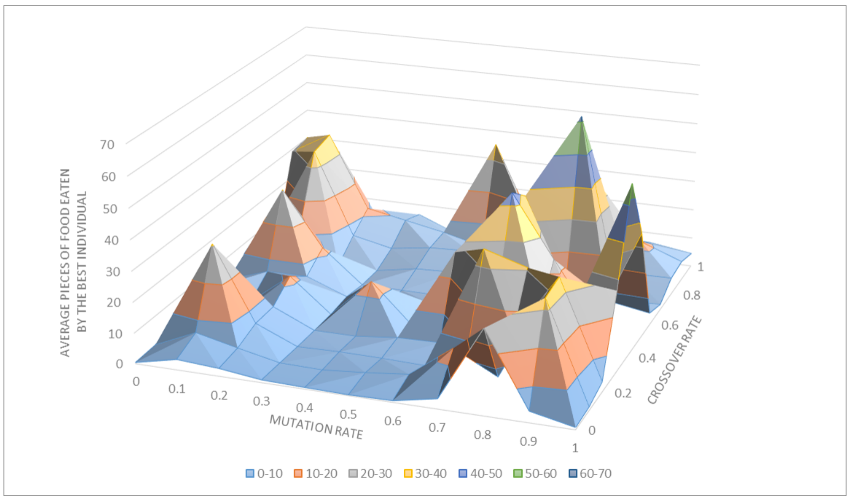

Figure 12: Average of 5 runs over the space of mutation and crossover rates, in increments of 0.1.

C Benchmark algorithm

Based on the design document by Ehlis [10].

```
import curses
import random
import operator
import numpy

from functools import partial

from deap import algorithms
from deap import base
from deap import creator
from deap import tools
from deap import gp
import multiprocessing

S_UP, S_RIGHT, S_DOWN, S_LEFT = 0, 1, 2, 3
XSIZE, YSIZE = 14, 14
INIT_SIZE = -1

# NOTE: YOU MAY NEED TO ADD A CHECK THAT THERE ARE ENOUGH SPACES LEFT FOR
# THE FOOD (IF THE TAIL IS VERY LONG)
NFOOD = 1
GENERATIONS = 150
POP = 1000
NUM_EVALS = 3
cxpb = 0.0
mutpb = 0.8

def if_then_else(condition, out1, out2):
    out1() if condition() else out2()

def progn(*args):
    for arg in args:
        arg()

def prog2(out1, out2):
    return partial(progn,out1,out2)

def prog3(out1, out2, out3):
    return partial(progn,out1,out2,out3)

# This class can be used to create a basic player object (snake agent)


class SnakePlayer(list):
    global S_RIGHT, S_LEFT, S_UP, S_DOWN
    global XSIZE, YSIZE
    global INIT_SIZE
    def __init__(self):
        self.direction = S_RIGHT
        self.body = [[4, 10], [4, 9], [4, 8], [4, 7], [4, 6],
                     [4, 5], [4, 4], [4, 3], [4, 2], [4, 1], [4, 0]]
        INIT_SIZE = len(self.body)
        self.score = 0
        self.ahead = []
        self.food = []

    def _reset(self):
        self.direction = S_RIGHT
        self.body[:] = [[4, 10], [4, 9], [4, 8], [4, 7], [4, 6],
                        [4, 5], [4, 4], [4, 3], [4, 2], [4, 1], [4, 0]]
        self.score = 0
        self.ahead = []
        self.food = []

    def is_tile_dangerous(self, tile):
        return (tile in self.body) or (tile[1]==0) or (tile[1]==XSIZE-1) or (tile[0]==0)
             or (tile[0]==YSIZE-1)

    def get_right_location(self):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
            tile[0]-=1
        elif self.direction == S_UP:
            tile[1]+=1
        elif self.direction == S_RIGHT:
            tile[0]+=1
        elif self.direction == S_DOWN:
            tile[1]-=1
        return tile

    def get_left_location(self):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
            tile[0]+=1
        elif self.direction == S_UP:
            tile[1]-=1
        elif self.direction == S_RIGHT:
            tile[0]-=1
        elif self.direction == S_DOWN:
            tile[1]+=1
        return tile

    def get_ahead_location(self):
        self.ahead = [ self.body[0][0] + (self.direction == S_DOWN and 1) + (self.
            direction == S_UP and -1), self.body[0][1] + (self.direction == S_LEFT and
            -1) + (self.direction == S_RIGHT and 1)]

    def get_ahead_2_location(self):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
            tile[1]-=2
        elif self.direction == S_UP:
            tile[0]-=2
        elif self.direction == S_RIGHT:
            tile[1]+=2
        elif self.direction == S_DOWN:
            tile[0]+=2
        return tile

    def updatePosition(self):
        self.get_ahead_location()
        self.body.insert(0, self.ahead)

    # You are free to define more sensing options to the snake

    def turn_left(self):
        self.direction = (self.direction - 1) % 4
    # You are free to define more sensing options to the snake

    def turn_right(self):
        self.direction = (self.direction + 1) % 4

    def go_straight(self):
        pass

    def snake_has_collided(self):
        self.hit = False
        if self.body[0][0] == 0 or self.body[0][0] == (
                    YSIZE - 1) or self.body[0][1] == 0 or self.body[0][1] == (XSIZE - 1)
                        :
            self.hit = True
        if self.body[0] in self.body[1:]:
            self.hit = True
        return (self.hit)

    def sense_danger_right(self):
        tile = self.get_right_location()
        return self.is_tile_dangerous(tile)

    def sense_danger_left(self):
        tile = self.get_left_location()
        return self.is_tile_dangerous(tile)

    def sense_danger_ahead(self):
        self.get_ahead_location()
        return self.is_tile_dangerous(self.ahead)

    def sense_danger_2_ahead(self):
        tile = self.get_ahead_2_location()
        return self.is_tile_dangerous(tile)

    def sense_wall_ahead(self):
        self.get_ahead_location()
        return( self.ahead[0] == 0 or self.ahead[0] == (YSIZE-1) or self.ahead[1] == 0
            or self.ahead[1] == (XSIZE-1) )

    def sense_food_ahead(self):
        self.get_ahead_location()
        return self.ahead in self.food

    def sense_tail_ahead(self):
        self.get_ahead_location()
        return self.ahead in self.body

    def sense_food_up(self):
        return self.food[0][0]<=self.body[0][0]

    def sense_food_right(self):
        return self.food[0][1]>=self.body[0][1]

    def sense_food_below(self):
        return not self.sense_food_up()

    def sense_food_left(self):
        return not self.sense_food_right()

    def sense_moving_up(self):
        return self.direction == S_UP

    def sense_moving_right(self):
        return self.direction == S_RIGHT

    def sense_moving_down(self):
        return self.direction == S_DOWN

    def sense_moving_left(self):
        return self.direction == S_LEFT

    def if_wall_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_wall_ahead, out1, out2)

    def if_food_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_food_ahead, out1, out2)

    def if_tail_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_tail_ahead, out1, out2)

    def if_danger_right(self, out1, out2):
        return partial(if_then_else, self.sense_danger_right, out1, out2)

    def if_danger_left(self, out1, out2):
        return partial(if_then_else, self.sense_danger_left, out1, out2)

    def if_danger_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_danger_ahead, out1, out2)

    def if_danger_2_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_danger_2_ahead, out1, out2)

    def if_food_up(self, out1, out2):
        return partial(if_then_else, self.sense_food_up, out1, out2)

    def if_food_right(self, out1, out2):
        return partial(if_then_else, self.sense_food_right, out1, out2)

    def if_food_left(self, out1, out2):
        return partial(if_then_else, self.sense_food_left, out1, out2)

    def if_food_down(self, out1, out2):
        return partial(if_then_else, self.sense_food_down, out1, out2)

    def if_moving_up(self, out1, out2):
        return partial(if_then_else, self.sense_moving_up, out1, out2)

    def if_moving_right(self, out1, out2):
```

```
        return partial(if_then_else, self.sense_moving_right, out1, out2)

    def if_moving_down(self, out1, out2):
        return partial(if_then_else, self.sense_moving_down, out1, out2)

    def if_moving_left(self, out1, out2):
        return partial(if_then_else, self.sense_moving_left, out1, out2)

def is_tile_empty(snake, food, tile):
    return not ((tile in snake.body) or (tile in food))

# This function places a food item in the environment
# TODO convert to using a spiral search around the point
def place_food(snake):
    food = []
    while len(food) < NFOOD:
        free_spaces = []
        for x in range(1, XSIZE - 2):
            for y in range(1, YSIZE - 2 ):
                if [x, y] not in snake and [x,y] not in food:
                    free_spaces.append([x,y])
        rand = random.randint(0, len(free_spaces)-1)
        food.insert(0, free_spaces[rand])
        # randx = random.randint(1, (XSIZE - 2))
        # randy = random.randint(1, (YSIZE - 2))
        # rand_food_tile = [randy, randx]


        # if is_tile_empty(snake, food, rand_food_tile):
        #     food.insert(0, rand_food_tile)
        # else:
        #     closest_free_tile = rand_food_tile
        #     min_d = -1
        #     for x in range(XSIZE-1):
        #         for y in range(YSIZE-1):
        #             # print x,y, is_tile_empty(snake, food, [y,x])
        #             if is_tile_empty(snake, food, [y,x]):
        #                 d = numpy.sqrt(numpy.power(y-randy,2) + numpy.power(x-randx,2)
            )
        #                 if d<min_d:
        #                     min_d = d
        #                     closest_free_tile[0] = y
        #                     closest_free_tile[1] = x
        #     food.insert(0, closest_free_tile)

    snake.food = food  # let the snake know where the food is
    return (food)



def runGame(individual):
    global snake
    global pset

    routine = gp.compile(individual, pset)

    total_score = 0
    for run in range(0,NUM_EVALS):
        snake._reset()
        food = place_food(snake)
        timer = 0


        while not snake.snake_has_collided() and not timer == XSIZE * YSIZE:
            ## EXECUTE THE SNAKE'S BEHAVIOUR HERE ##
            if len(food)<NFOOD:
                food = place_food(snake)
            routine()
            snake.updatePosition()

            if snake.body[0] in food:
                snake.score += 1
                timer = 0
            else:
                snake.body.pop()
                timer += 1  # timesteps since last eaten
        if snake.score==0:
            snake.score = -(abs(snake.body[0][0] - food[0][0]) + abs(snake.body[0][1] -
                food[0][1]))
        total_score += snake.score
    return total_score/NUM_EVALS,

def run_debug(individual):
    global snake
    global pset

    routine = gp.compile(individual, pset)

    total_score = 0
    for run in range(0,NUM_EVALS):
        snake._reset()
        food = place_food(snake)
        timer = 0

        while not snake.snake_has_collided() and not timer == XSIZE * YSIZE:
            ## EXECUTE THE SNAKE'S BEHAVIOUR HERE ##
            print snake.direction, snake.body[0], snake.sense_danger_ahead()
            routine()
            print snake.direction
            snake.updatePosition()
            if snake.body[0] in food:
                snake.score += 1
                food = place_food(snake)
                timer = 0
            else:
                snake.body.pop()
                timer += 1  # timesteps since last eaten
        if snake.score==0:
            snake.score = -(abs(snake.body[0][0] - food[0][0]) + abs(snake.body[0][1] -
                food[0][1]))
        total_score += snake.score


    collided = snake.snake_has_collided()
    hitBounds = snake.body[0][0] == 0 or snake.body[0][0] == (
                    YSIZE - 1) or snake.body[0][1] == 0 or snake.body[0][1] == (XSIZE -
                        1)
    print "Collided:␣", collided
    print "Hit␣wall:␣", hitBounds
    return total_score/NUM_EVALS,

# This outline function is the same as runGame (see below). However,
# it displays the game graphically and thus runs slower
# This function is designed for you to be able to view and assess
# your strategies, rather than use during the course of evolution
def displayStrategyRun(individual):

    routine = gp.compile(individual, pset)

    curses.initscr()
    win = curses.newwin(YSIZE, XSIZE, 0, 0)
    win.keypad(1)
    curses.noecho()
    curses.curs_set(0)
    win.border(0)
    win.nodelay(1)
    win.timeout(120)

    snake._reset()
    food = place_food(snake)

    for f in food:
        win.addch(f[0], f[1], '@')

    timer = 0
    collided = False
    tmp = []
    while not collided and not timer == ((2 * XSIZE) * YSIZE):
        # Set up the display
        win.border(0)
        win.addstr(0, 2, 'Score␣:␣' + str(snake.score) + '␣')
        win.getch()
        ## EXECUTE THE SNAKE'S BEHAVIOUR HERE ##
        routine()
        snake.updatePosition()

        if snake.body[0] in food:
            snake.score += 1
            for f in food:
                win.addch(f[0], f[1], '␣')
            food = place_food(snake)
            for f in food:
                win.addch(f[0], f[1], '@')
            timer = 0
        else:
            last = snake.body.pop()
            win.addch(last[0], last[1], '␣')
            timer += 1  # timesteps since last eaten
        win.addch(snake.body[0][0], snake.body[0][1], 'o')

        collided = snake.snake_has_collided()
        hitBounds = snake.body[0][0] == 0 or snake.body[0][0] == (
                    YSIZE - 1) or snake.body[0][1] == 0 or snake.body[0][1] == (XSIZE -
                        1)
    curses.endwin()
    print "Collided:␣", collided
    print "Hit␣wall:␣", hitBounds
    print "Score:␣", snake.score
    raw_input("Press␣to␣continue...")

    return snake.score,


def main():
    global snake
    global pset

    snake = SnakePlayer()

    pset = gp.PrimitiveSet("MAIN", 0)
    # pset.addPrimitive(operator.and_, 2, name="aNd")
    # pset.addPrimitive(operator.or_, 2, name="Or")
    # pset.addPrimitive(operator.not_, 2, name="not")
    # pset.addPrimitive(prog2, 2)
    # pset.addPrimitive(prog3, 3)

    pset.addPrimitive(snake.if_food_ahead, 2, name="if_food_ahead")
    pset.addPrimitive(snake.if_danger_ahead, 2, name="if_danger_ahead")
    pset.addPrimitive(snake.if_danger_right, 2, name="if_danger_right")
    pset.addPrimitive(snake.if_danger_left, 2, name="if_danger_left")
    pset.addPrimitive(snake.if_danger_2_ahead, 2, name="if_danger_2_ahead")
    pset.addPrimitive(snake.if_food_up, 2, name="if_food_up")
    pset.addPrimitive(snake.if_food_right, 2, name="if_food_right")
    pset.addPrimitive(snake.if_moving_up, 2, name="if_moving_up")
    pset.addPrimitive(snake.if_moving_right, 2, name="if_moving_right")
    pset.addPrimitive(snake.if_moving_down, 2, name="if_moving_down")
    pset.addPrimitive(snake.if_moving_left, 2, name="if_moving_left")



    pset.addTerminal(snake.turn_right, name="turn_right")
    pset.addTerminal(snake.turn_left, name="turn_left")
    pset.addTerminal(snake.go_straight, name="go_straight")

    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()


    pool = multiprocessing.Pool()
    toolbox.register("map", pool.map)


    toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=2, max_=4)
    toolbox.register("individual", tools.initIterate,
                     creator.Individual, toolbox.expr)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)
    toolbox.register("compile", gp.compile, pset=pset)
    toolbox.register("evaluate", runGame)
    toolbox.register("select", tools.selDoubleTournament, fitness_size=2, parsimony_size
        =1.15, fitness_first=True)
    toolbox.register("mate", gp.cxOnePoint)
    toolbox.register("expr_mut", gp.genHalfAndHalf, min_=1, max_=3, pset=pset)
    toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)

    toolbox.decorate("mate", gp.staticLimit(
        key=operator.attrgetter("height"), max_value=15))
    toolbox.decorate("mutate", gp.staticLimit(
        key=operator.attrgetter("height"), max_value=15))


    stats_fit = tools.Statistics(lambda ind: ind.fitness.values)
    stats_size = tools.Statistics(len)
    mstats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
    mstats.register("avg", lambda x: float(int(numpy.mean(x)*100))/100)
    mstats.register("std", lambda x: float(int(numpy.std(x)*100))/100)
    # mstats.register("min", numpy.min)
    mstats.register("max", numpy.max)



    ## THIS IS WHERE YOUR CORE EVOLUTIONARY ALGORITHM WILL GO #
    random.seed()
    pop = toolbox.population(n=POP)
    hof = tools.HallOfFame(5)
    try:
        pop, log = algorithms.eaSimple(
            pop, toolbox, cxpb , mutpb, GENERATIONS, stats=mstats, halloffame=hof,
                verbose=True)

    except KeyboardInterrupt:
        pool.terminate()
        pool.join()
        raise KeyboardInterrupt
    return mstats.compile(pop), tools.selBest(pop, 1)[0]


if __name__ == "__main__":
    try:
        for x in range(0,28):
            out = main()
            record = out[0]
            expr = str(out[1])
            row = (record['fitness']['avg'], record['fitness']['max'], record['fitness'
                ]['std'], record['size']['avg'], record['size']['max'], record['size']['
                std'], "\r")
            fd = open('approach1results.csv', 'a')
            fd.write(",".join(map(str, row)))
            fd.close()
            if runGame(expr)[0]>=133:
                fd = open('snakesbest.txt', 'a')
            else:
                fd = open('snakes.txt', 'a')
            fd.write(expr)
            fd.close()
    except KeyboardInterrupt:
        print "Terminated␣by␣user,␣after␣%s␣iterations" % str(x)
```

D New algorithm

Based on the design document by Ehlis [10].

```
# MutShrink with low mutation rate
import curses
import random
import operator
import numpy

from functools import partial
from collections import deque

# import algorithms
from deap import algorithms
from deap import base
from deap import creator
from deap import tools
from deap import gp
# import pygraphviz as pgv
import multiprocessing
import sys
import argparse as ap

S_UP, S_RIGHT, S_DOWN, S_LEFT = 0, 1, 2, 3
XSIZE, YSIZE = 14, 14
INIT_SIZE = -1

# NOTE: YOU MAY NEED TO ADD A CHECK THAT THERE ARE ENOUGH SPACES LEFT FOR
# THE FOOD (IF THE TAIL IS VERY LONG)
NFOOD = 1
GENERATIONS = 90
POP = 1000
NUM_EVALS = 1
cxpb = 0.7
mutpb = 0.8
parsimony = 1.05
max_tree_depth = 7

def if_then_else(condition, out1, out2):
    out1() if condition() else out2()

# This class can be used to create a basic player object (snake agent)

class SnakePlayer(list):
    global S_RIGHT, S_LEFT, S_UP, S_DOWN
    global XSIZE, YSIZE
    global INIT_SIZE
    def __init__(self):
        self.direction = S_RIGHT
        self.body = deque([[4, 10], [4, 9], [4, 8], [4, 7], [4, 6],
                     [4, 5], [4, 4], [4, 3], [4, 2], [4, 1], [4, 0]])
        INIT_SIZE = len(self.body)
        self.initial_size = len(self.body)
        self.score = 0
        self.ahead = []
        self.food = []

    def _reset(self):
        self.direction = S_RIGHT
        self.body = deque([[4, 10], [4, 9], [4, 8], [4, 7], [4, 6],
                     [4, 5], [4, 4], [4, 3], [4, 2], [4, 1], [4, 0]])
        self.score = 0
        self.initial_size = len(self.body)
        self.ahead = []
        self.food = []

    def is_tile_dangerous(self, tile):
        return (tile in self.body) or (tile[1]==0) or (tile[1]==XSIZE-1) or (tile[0]==0)
             or (tile[0]==YSIZE-1)

    def get_right_location(self, distance):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
            tile[0]-=distance
        elif self.direction == S_UP:
            tile[1]+=distance
        elif self.direction == S_RIGHT:
            tile[0]+=distance
        elif self.direction == S_DOWN:
            tile[1]-=distance
        return tile

    def get_left_location(self, distance):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
            tile[0]+=distance
        elif self.direction == S_UP:
            tile[1]-=distance
        elif self.direction == S_RIGHT:
            tile[0]-=distance
        elif self.direction == S_DOWN:
            tile[1]+=distance
        return tile

    def get_ahead_location(self):
        self.ahead = [ self.body[0][0] + (self.direction == S_DOWN and 1) + (self.
            direction == S_UP and -1), self.body[0][1] + (self.direction == S_LEFT and
            -1) + (self.direction == S_RIGHT and 1)]

    def get_ahead_2_location(self):
        tile = [self.body[0][0], self.body[0][1]]
        if self.direction == S_LEFT:
            tile[1]-=2
        elif self.direction == S_UP:
            tile[0]-=2
        elif self.direction == S_RIGHT:
            tile[1]+=2
        elif self.direction == S_DOWN:
            tile[0]+=2
        return tile

    def updatePosition(self):
        self.get_ahead_location()
        self.body.appendleft(self.ahead)

    # You are free to define more sensing options to the snake
    def turn_left(self):
        self.direction = (self.direction - 1) % 4

    # You are free to define more sensing options to the snake
    def turn_right(self):
        self.direction = (self.direction + 1) % 4

    def go_straight(self):
        pass

    def changeDirectionUp(self):
        self.direction = S_UP

    def changeDirectionRight(self):
        self.direction = S_RIGHT

    def changeDirectionDown(self):
        self.direction = S_DOWN

    def changeDirectionLeft(self):
        self.direction = S_LEFT

    def snake_has_collided(self):
        self.hit = False
        if self.body[0][0] == 0 or self.body[0][0] == (
                    YSIZE - 1) or self.body[0][1] == 0 or self.body[0][1] == (XSIZE - 1)
                        :
            self.hit = True
        # for i, val in enumerate(self.body):
        #     if (not i) and (self.body[0]==val):
        #         self.hit = True
        #         break
        for x in range (1, len(self.body)):
            if self.body[x] == self.body[0]:
                self.hit = True
                break
        # if self.body[0] in list(self.body)[1:]:
        #     self.hit = True
        return (self.hit)

    def sense_wall_ahead(self):
        self.get_ahead_location()
        return( self.ahead[0] == 0 or self.ahead[0] == (YSIZE-1) or self.ahead[1] == 0
            or self.ahead[1] == (XSIZE-1) )

    def sense_food_ahead(self):
        self.get_ahead_location()
        return self.ahead in self.food

    def sense_wall_ahead(self):
        self.get_ahead_location()
        return( self.ahead[0] == 0 or self.ahead[0] == (YSIZE-1) or self.ahead[1] == 0
            or self.ahead[1] == (XSIZE-1) )

    def sense_wall_right(self):
        tile = self.get_right_location(1)
        return( tile[0] == 0 or tile[0] == (YSIZE-1) or tile[1] == 0 or tile[1] == (
            XSIZE-1) )

    def sense_wall_left(self):
        tile = self.get_left_location(1)
        return( tile[0] == 0 or tile[0] == (YSIZE-1) or tile[1] == 0 or tile[1] == (
            XSIZE-1) )

    def sense_wall_2_ahead(self):
        tile = self.get_ahead_2_location()
        return( tile[0] == 0 or tile[0] == (YSIZE-1) or tile[1] == 0 or tile[1] == (
            XSIZE-1) )

    def sense_moving_up(self):
        return self.direction == S_UP

    def sense_moving_right(self):
        return self.direction == S_RIGHT

    def sense_moving_down(self):
        return self.direction == S_DOWN

    def sense_moving_left(self):
        return self.direction == S_LEFT

    def sense_wall_2_away(self):
        tile = self.body[0]
        return (tile[0] in [2, XSIZE-2] or tile[1] in [2, XSIZE-2])

    def if_wall_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_wall_ahead, out1, out2)

    def if_wall_right(self, out1, out2):
        return partial(if_then_else, self.sense_wall_right, out1, out2)

    def if_wall_left(self, out1, out2):
        return partial(if_then_else, self.sense_wall_left, out1, out2)

    def if_wall_2_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_wall_2_ahead, out1, out2)

    def if_wall_2_left(self, out1, out2):
        return partial(if_then_else, self.sense_wall_2_left, out1, out2)

    def if_food_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_food_ahead, out1, out2)

    def if_tail_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_tail_ahead, out1, out2)

    def if_moving_up(self, out1, out2):
        return partial(if_then_else, self.sense_moving_up, out1, out2)

    def if_moving_right(self, out1, out2):
        return partial(if_then_else, self.sense_moving_right, out1, out2)
    def if_moving_down(self, out1, out2):
        return partial(if_then_else, self.sense_moving_down, out1, out2)

    def if_moving_left(self, out1, out2):
        return partial(if_then_else, self.sense_moving_left, out1, out2)

    def if_wall_2_away(self, out1, out2):
        return partial(if_then_else, self.sense_wall_2_away, out1, out2)

def is_tile_empty(snake, food, tile):
    return not ((tile in snake.body) or (tile in food))

# This function places a food item in the environment
# TODO convert to using a spiral search around the point
def place_food(snake):
    food = []

    while len(food) < NFOOD:
        free_spaces = []
        for x in range(1, XSIZE - 1):
            for y in range(1, YSIZE - 1):
                if [x, y] not in snake.body and [x,y] not in food:
                    free_spaces.append([x,y])
        if len(free_spaces)==0:
            return ()
        rand = random.randint(0, len(free_spaces)-1)
        food.insert(0, free_spaces[rand])

    snake.food = food  # let the snake know where the food is
    return (food)


def runGame(individual):
    global snake
    global pset

    routine = gp.compile(individual, pset)

    total_score = 0.0
    total_steps = 0.0
    for run in range(0,NUM_EVALS):
        snake._reset()
        food = place_food(snake)
        timer = 0

        tour = set()
        tours = 0
        steps = 0
        while not snake.snake_has_collided() and not timer == (XSIZE-2) * (YSIZE-2):
            # Verify game-winning condition
            if snake.score == ((XSIZE-2) * (YSIZE-2)) - snake.initial_size:
                break
            routine()
            snake.updatePosition()
            if snake.body[0] in food:
                snake.score += 1
                try:
                    food = place_food(snake)
                except ValueError:
                    break
                timer = 0
            else:
                snake.body.pop()
                timer += 1  # timesteps since last eaten
            steps+=1
            tour.add(str(snake.body[0]))
        # Penalise snakes which have timed out
        if timer == XSIZE * YSIZE:
            return -10,
        total_steps += steps
        total_score += snake.score

    avg_steps = total_steps/(NUM_EVALS*100)
    avg_score = total_score/NUM_EVALS
    coverage = float(len(tour))/float((XSIZE-2)*(YSIZE-2))
    return coverage + avg_steps,
def runInGame(individual, evals):
    global snake
    global pset

    routine = gp.compile(individual, pset)

    total_score = 0.0
    for run in range(0, evals):
        snake._reset()
        food = place_food(snake)
        timer = 0

        while not snake.snake_has_collided() and not timer == (XSIZE-2) * (YSIZE-2):
            routine()
            snake.updatePosition()


            if snake.body[0] in food:
                snake.score += 1
                try:
                    food = place_food(snake)
                except ValueError:
                    break
                timer = 0
            else:
                snake.body.pop()
                timer += 1  # timesteps since last eaten
        total_score += snake.score
    avg_score = total_score/evals
    return avg_score



# This outline function is the same as runGame (see below). However,
# it displays the game graphically and thus runs slower
# This function is designed for you to be able to view and assess
# your strategies, rather than use during the course of evolution
def displayStrategyRun(individual):

    routine = gp.compile(individual, pset)

    curses.initscr()
    win = curses.newwin(YSIZE, XSIZE, 0, 0)
    win.keypad(1)
    curses.noecho()
    curses.curs_set(0)
    win.border(0)
    win.nodelay(1)
    win.timeout(60)

    snake._reset()
    food = place_food(snake)

    for f in food:
        win.addch(f[0], f[1], '@')
    for b in snake.body:
        win.addch(b[0], b[1], 'o')

    timer = 0
    collided = False
    tmp = []
    while not collided and not timer == ((2 * XSIZE) * YSIZE):
        # Set up the display
        win.border(0)
        win.addstr(0, 2, 'Score␣:␣' + str(snake.score) + '␣')
        win.getch()

        ## EXECUTE THE SNAKE'S BEHAVIOUR HERE ##
        routine()
        snake.updatePosition()

        if snake.body[0] in food:
            snake.score += 1
            for f in food:
                win.addch(f[0], f[1], '␣')
            food = place_food(snake)
```

```
                timer = 0
            else:
                last = snake.body.pop()
                win.addch(last[0], last[1], '␣')
                timer += 1  # timesteps since last eaten
            win.addch(snake.body[0][0], snake.body[0][1], 'o')
            for f in food:
                win.addch(f[0], f[1], '@')

            collided = snake.snake_has_collided()
            hitBounds = snake.body[0][0] == 0 or snake.body[0][0] == (
                        YSIZE - 1) or snake.body[0][1] == 0 or snake.body[0][1] == (XSIZE -
                            1)

        curses.endwin()
        print "Collided:␣", collided
        print "Hit␣wall:␣", hitBounds
        print "Score:␣", snake.score
        raw_input("Press␣to␣continue...")

        return snake.score,


def main(multicore, seeded):
    global snake
    global pset
    snake = SnakePlayer()

    pset = gp.PrimitiveSet("MAIN", 0)

    pset.addPrimitive(snake.if_wall_2_away, 2, name="if_wall_2_away")
    pset.addPrimitive(snake.if_wall_left, 2, name="if_wall_left")
    pset.addPrimitive(snake.if_wall_ahead, 2, name="if_wall_ahead")
    pset.addPrimitive(snake.if_wall_right, 2, name="if_wall_right")

    pset.addPrimitive(snake.if_moving_up, 2, name="if_moving_up")
    pset.addPrimitive(snake.if_moving_right, 2, name="if_moving_right")
    pset.addPrimitive(snake.if_moving_down, 2, name="if_moving_down")
    pset.addPrimitive(snake.if_moving_left, 2, name="if_moving_left")

    pset.addTerminal(snake.changeDirectionUp, name="go_up")
    pset.addTerminal(snake.changeDirectionDown, name="go_down")
    pset.addTerminal(snake.changeDirectionLeft, name="go_left")
    pset.addTerminal(snake.changeDirectionRight, name="go_right")
    pset.addTerminal(snake.go_straight, name="go_straight")

    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()
    toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=2, max_=6)
    toolbox.register("individual", tools.initIterate,
                     creator.Individual, toolbox.expr)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)
    toolbox.register("compile", gp.compile, pset=pset)

    toolbox.register("evaluate", runGame)
    toolbox.register("select", tools.selDoubleTournament, fitness_size=5, parsimony_size
        =parsimony, fitness_first=True)
    toolbox.register("mate", gp.cxOnePoint)
    toolbox.register("mutate", gp.mutNodeReplacement, pset=pset)

    toolbox.decorate("mate", gp.staticLimit(
        key=operator.attrgetter("height"), max_value=max_tree_depth))
    toolbox.decorate("mutate", gp.staticLimit(
        key=operator.attrgetter("height"), max_value=max_tree_depth))

    stats_fit = tools.Statistics(lambda ind: ind.fitness.values)
    stats_size = tools.Statistics(len)
    mstats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
    mstats.register("avg", lambda x: float(int(numpy.mean(x)*100))/100)
    mstats.register("std", lambda x: float(int(numpy.std(x)*100))/100)
    mstats.register("max", numpy.max)

    if multicore:
        pool = multiprocessing.Pool()
        toolbox.register("map", pool.map)
    pop = toolbox.population(n=POP)
    hof = tools.HallOfFame(3)
    try:
        pop, log = algorithms.eaSimple(
            pop, toolbox, cxpb, mutpb, GENERATIONS, stats=mstats, halloffame=hof,
                verbose=True)
        expr = tools.selBest(pop, 1)[0]
        print expr


        evals = 100
        val = runInGame(expr, evals)
        print "Evaluating:␣", str(evals), "times,␣average␣score:␣", str(val)

        if seeded:
            displayStrategyRun(expr)

    except KeyboardInterrupt:
        if multicore:
            pool.terminate()
            pool.join()
        raise KeyboardInterrupt
    return mstats.compile(pop), val

if __name__ == "__main__":

    parser = ap.ArgumentParser(description="My␣Script")
    parser.add_argument("--iterations", type=int)
    parser.add_argument("--multicore", action='store_true')
    parser.add_argument("--seed")
    parser.add_argument("--save_results", action='store_true')
    parser.add_argument("--max_gen", type=int)


    args, leftovers = parser.parse_known_args()

    iterations = 1 if args.iterations is None else int(args.iterations)
    multicore = args.multicore

    if args.seed is not None:
        iterations = 1
        multicore = False
        print "Seed␣detected,␣ignoring␣other␣flags␣(running␣single␣run␣on␣non-multicore␣
            process␣to␣ensure␣deterministic␣execution)."

    if args.max_gen is not None:
        GENERATIONS = args.max_gen
    try:
        for x in range(0, iterations):
            if args.seed is not None:
                seed = float(args.seed)
            else:
                seed = random.random()
            random.seed(seed)
            out = main(multicore, args.seed is not None)
            record = out[0]
            if args.save_results:
                row = (record['fitness']['avg'], record['fitness']['max'], record['
                    fitness']['std'], record['size']['avg'], record['size']['max'],
                    record['size']['std'], out[1], "\r")
                fd = open('approach4_results.csv', 'a')
                fd.write(",".join(map(str, row)))
                fd.close()
    except KeyboardInterrupt:
        print "Terminated␣by␣user,␣after␣%s␣iterations" % str(x)
```