AIML402 Assignment 2

By Hugo Phibbs

Introduction

This report details the development of a genetic algorithm (GA) to develop a population of 'smart' cleaners.

A population of cleaners is guided by this GA to compete with another opponent population to suck up dust from squares within a grid area.

Within the grid, a single cleaner can move around, suck up dust from squares, empty its dust bin (a cleaner can hold a finite amount of dust), and recharge its battery energy.

The goal of a cleaner population is to maximise the number of dust loads picked up, while at the same time minimising the amount of dust picked up by the rival cleaner population.

The Algorithm Explained

I will provide a high-level overview of the algorithm here, later I will discuss and justify the design of the model.

Chromosomes

Chromosome Structure

Each cleaner has an attached chromosome, this is the basis of the GA described in this report.

A chromosome is split into weights and biases, each treated independently. This allows for greater flexibility when experimenting with different techniques to crossover bias and weight values.

Put simply, a chromosome is a combination of a matrix of weights W, and a vector of biases $\boldsymbol{b}$. A given chromosome C can thus be written as:

$$C = (W, \boldsymbol{b})$$

Where:

- W has dimensions of $4 * 63$ - 4 rows for the four possible actions, and 63 columns for weight values for each percept value (see below). One can break down W into 2 submatrices, a weight matrix for the visual percepts W_V, and another for the current state of the cleaner $W_{\boldsymbol{s}}$. Hence write W as.

$$W = [W_V, W_s]$$

- $\boldsymbol{b}$ is a 4 dimensional vector.

I will explain the range of weight and bias values once it is clear how cleaners decide on what action to take.

Finding actions

At a given point in the game, a cleaner decides on what action to take – either move forward/backwards or turn left/right – by expressing its preference for each of these options in the action vector $\boldsymbol{a}$.

To calculate $\boldsymbol{a}$, it receives a set of percepts from its current state in the game.

These percepts include:

- V , a $3 * 5 * 4 = 60$ dimensional matrix of its visual field
- $battery$, the current level of the cleaner's battery.
- bin, the current number of free spots left in the cleaner's bin.
- $fails$, the total number of failed actions of the cleaner

Combine the latter 3 percepts into the vector $\boldsymbol{s} = [battery, bin, fails]$.

The algorithm then flattens these percepts into a 63 dimensional vector $\boldsymbol{v} = flatten(V) \ + \boldsymbol{s}$.

Finally, the action vector $\boldsymbol{a}$ calculated according to the formula:

$$\boldsymbol{a} = W\boldsymbol{v} + W_s^2\boldsymbol{s} + \boldsymbol{b} \quad (\boldsymbol{F})$$

Besides the $W_s^2\boldsymbol{s}$ term, $\boldsymbol{F}$ is linear with respect to weight and biases values (resembling a single linear perceptron). I considered entries in $\boldsymbol{s}$ relative relatively more important than any entries in V, hence I added the $W_s^2\boldsymbol{s}$ term to increase the effective weighting of the entries of $\boldsymbol{s}$.[1]

A given weight value $W_{i,j}$ value is chosen from $U(0, 10)$. To balance possibly large values of $W\boldsymbol{v} + W_s^2\boldsymbol{s}$ a bias value $\boldsymbol{b_i}$ is taken from $U(0, 10 * \#percepts) = U(0, 630)$[2].

The Genetic Algorithm

Here, I will briefly explain the key aspects of the genetic algorithm, and how they are implemented. Later the analysis section justifies these choices.

In a nutshell, this genetic algorithm can be broken down into fitness evaluation, parent selection, chromosome crossover, mutation, and population replacement – let's explore these one by one.

Fitness evaluation

The algorithm uses the standard fitness that counts the number of tiles cleaned by a cleaner during a round. Formally:

$$f = \#cleaned$$

Parent selection

Tournament selection was used to select parents.

This was done by creating a subset with a size of 20% of the current population size, then choosing the top 2 parents based on fitness to create a child from.

[1] This term's effect on performance may in-fact be marginal.

[2] Through ad hoc testing, I larger values of biases to be detrimental on performance. Due to scope of this report, I decided not to go in-depth on bias values.

A subset proportion of 20% was to sway chances of the subset containing good parents, while not ruling out the possibility of containing poor parents – thus encouraging greater population diversity.

Chromosome Crossover

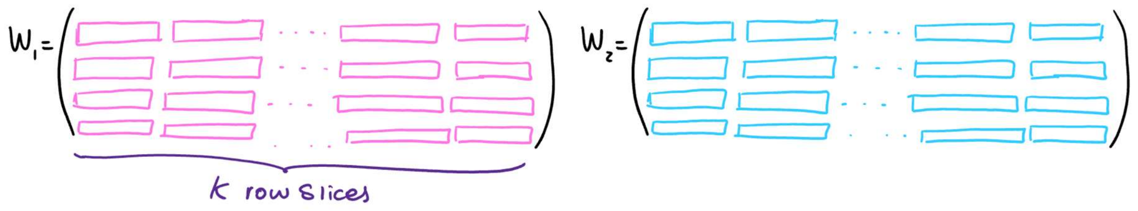
The algorithm's crossover technique of two parent chromosomes $C_1(W_1, \boldsymbol{b}_1)$ and $C_2(W_2, \boldsymbol{b}_2)$ creates a child $C(W, \boldsymbol{b})$. This process combines the weight matrices and bias vectors separately.

Weight crossover

K-point crossover was chosen to crossover two weight matrices W_1 and W_2

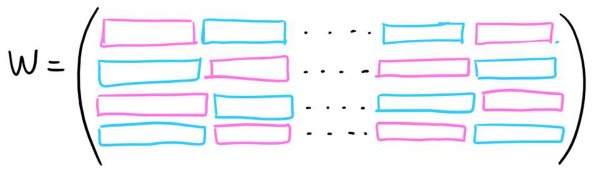
This is done by creating k even length slices of the rows of W_1 and W_2, then interleaving them together in an alternating fashion (in a checkerboard fashion) to create W.

To explain this, consider the sketches below of W_1 and W_2 – I've colour coded their k-slices for clarity.



Sketch: Segmenting rows of W_1 and W_2 into k slices

W_1 and W_2 are then combined to create W, which has a structure like the sketch below:



Sketch: How a matrix W is created from segments from W_1 and W_2

For the rest of the report, and for the submission, assume $k = 4$.

Bias crossover

I used linear interpolation of bias values. A new bias vector $\boldsymbol{b}$ is created from two bias vectors $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ with the formula:

$$\boldsymbol{b} = \boldsymbol{b}_1 \boldsymbol{\alpha} + (\boldsymbol{1} - \boldsymbol{\alpha})\boldsymbol{b}_2$$

Where $\boldsymbol{\alpha}$ is a 4 dimensional vector with entries from $U(0,1)$.

Mutation

Mutation of chromosomes works by randomly changing the values of the weight matrix W. To perform this, each row of W has a 1% chance of being mutated. If a row is then chosen to be mutated, a random entry in this row is set to random value from the range R, given as:

$$R = (\mathrm{minWeight} - \frac{\mathrm{maxMinDist}}{4}, maxWeight + \frac{maxMinDist}{4})$$

Where:

$$minWeight = \min(W), maxWeight = \max(W)$$

$$maxMinDist = \frac{maxWeight + minWeight}{2}$$

The range R was chosen as such to choose a potentially familiar weight value between $minWeight$ and $maxWeight$, while at the same time having the chance to choose a (possibly beneficial) novel value.

Population replacement

Instead of creating a new population every generation (as per the standard approach), I averaged the fitness of individual cleaners across 10 generations.

From here, the cleaner population is ranked according to average fitness. Elitism is then used to preserve the top 20% of cleaners for the next population.

Children cleaners are then created from the entire population, to keep population size constant, create k new children, where k is given by:

$$k = populationSize - \ numEliteCleaners$$
$$= populationSize - floor(0.2 * populationSize)$$

In summary, a new population is created from a set of new children, along with the elite cleaners from the previous one.

Discussion, Justifications and Results

This section explores a few different choices of model parameters and characteristics, in the goal to justify the configuration of the model.

Notes on this section

It should be noted that results in the analysis section should consider the effects of experimental error. While one technique may appear excellent over on trial, it may prove mediocre in subsequent trials (and vice versa). However, due to time constraints, I couldn't fully explore different technique using very large trials.

Due to practicalities of time, during analysis, I kept the number of generations of cleaners while cleaning to 300 (150 against random, and 150 against self). Besides, I found average population fitness often plateaued well before the 300th generation.

Fitness function

Due to time constraints, I couldn't explore alternative fitness functions. Consequently, I kept with the standard fitness function that counts the number of tiles cleaned.

Evaluating cleaner fitness

To impart greater stability and potentially better performance during the training of the cleaners, I created children's chromosomes every 10^{th} generation. To rank the chromosome population fitness, I averaged each chromosomes fitness over the previous 10 generations.

To illustrate the benefit of this technique over the default, see the graph below.

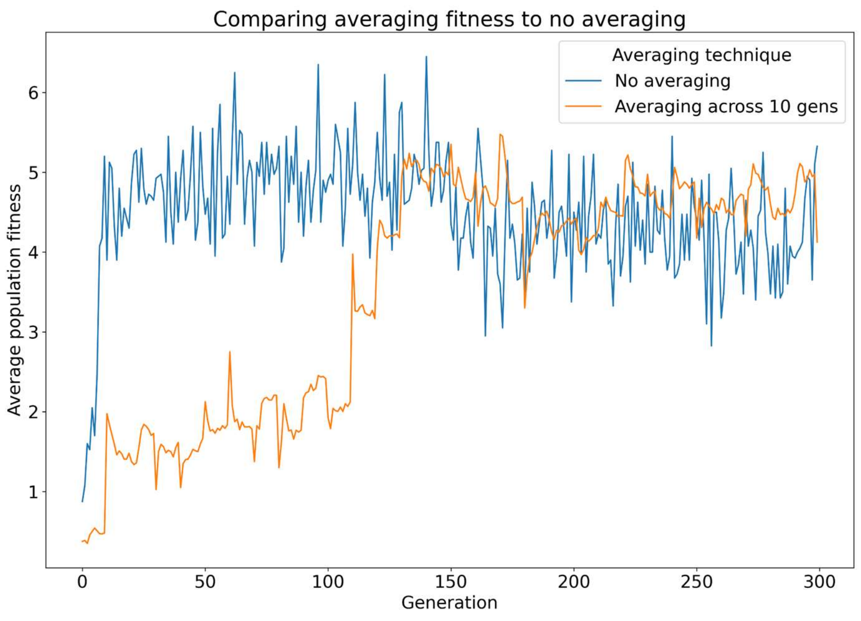


Figure: Comparing averaging of cleaner fitness over 10 generations to no averaging.
Done across 300 generations using random uniform crossover.

There is a lot of 'noise' when the model trains with no averaging, making comparisons between model techniques difficult – e.g., it could be hard to pick a clear winner between k-point and random uniform crossover if the graphs are so jagged.

Another advantage of averaging is that it gives the chance for good chromosomes to prove their worth over several generations. Judging chromosome fitness over just one generation is susceptible to random fluctuations due to pure chance, behaviour of other cleaners etc.

What is also remarkable is that the no-averaging technique jumps up in average fitness very quickly. This was most likely a fluke due to a chance occurrence of excellent weight values for the population very early on.

For the rest of the report, unless otherwise stated, assume that the model averages chromosome performance across 10 generations.

Crossover strategy

I experimented with multiple different strategies for the crossover of weights.

Weights

In developing this model, I experimented with k-point, random uniform, and linear interpolation crossover of the matrix W. I'll briefly explain my implementation of the latter two, then compare them together. To avoid repeating myself, assuming that you are performing a cross over of two parent weight matrices $W_1$and W_2, to create a child weight matrix W.

Linear Interpolation

W is simply given by the formula:

$$W = W \circ A + W \circ (\mathbf{1}_{6*63} - A)$$

Where ∘ is the Hadaman (elementwise) matrix product, and the $4 * 63$ dimensional matrix A is a matrix containing elements from the unform distribution $U(0,1\ )$, formally:

$$A \sim U_{4*63}(0,1)$$

Random Uniform

This was simply element wise random crossover between W_1 and W_2. Expressed as a formula, you can build W elementwise with the formula:

$$W_{i,j} = \begin{cases} W_{1\ i,j} & if\ \alpha < 0.5 \\ W_{2\ i,j} & if\ \alpha \geq 0.5 \end{cases}$$

Where i and j are row and column indices respectively, and α is chosen from $U(0,1)$.

Comparing crossover techniques

The graph below shows the average population fitness using different crossover techniques. All crossover techniques seem to converge to a fitness of around 4.5-5 (Meaning on average 4.5-5 tiles cleaned).

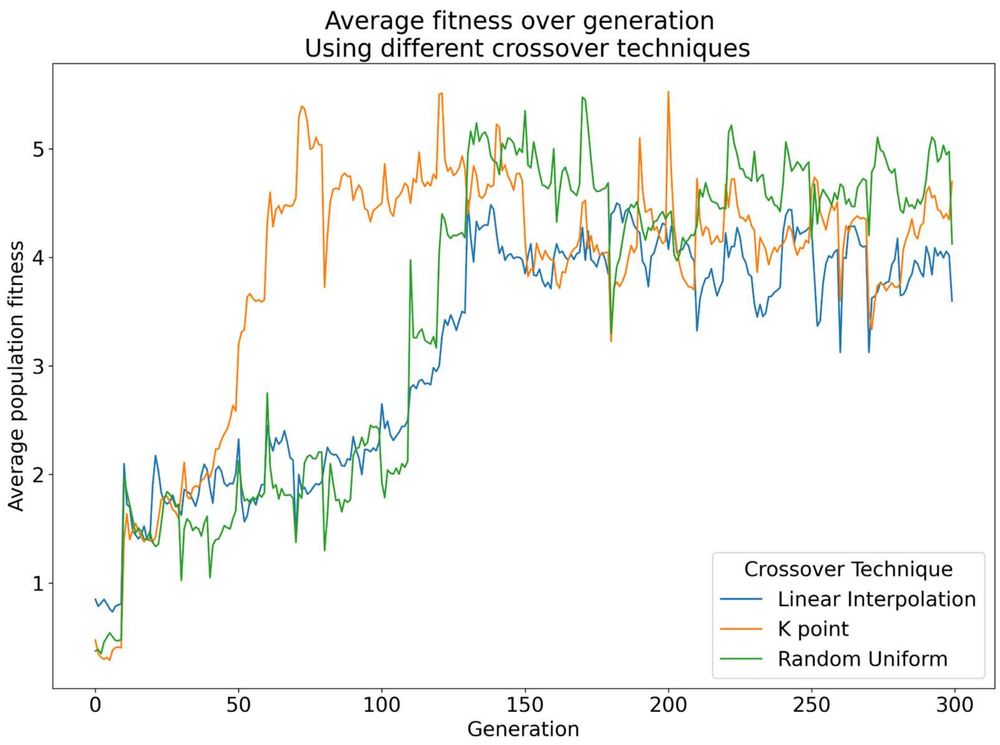


Figure: Comparing the average population fitness of crossover techniques across 300 generations

What was surprising was that the performance of each crossover technique was comparable - there was no clear winner.

However, relatively speaking, k-point converges relatively fast to its average performance. This could be due to k-point preserving specific patterns in the chromosome that are rewarded with high fitness – while other techniques may scramble these patterns, thus decreasing overall population fitness. For this reason, I chose k-point crossover for the final submission.

Parent selection

The chosen strategy for parent selection was tournament selection. However, I also experimented with Roulette selection.

Roulette selection

This was done by following the algorithm from the lecture notes. As such, I don't think it's necessary to reiterate how roulette selection works.

Comparing selection techniques

The graph below compares the average fitness across 300 generations for two populations created with roulette and tournament selection.

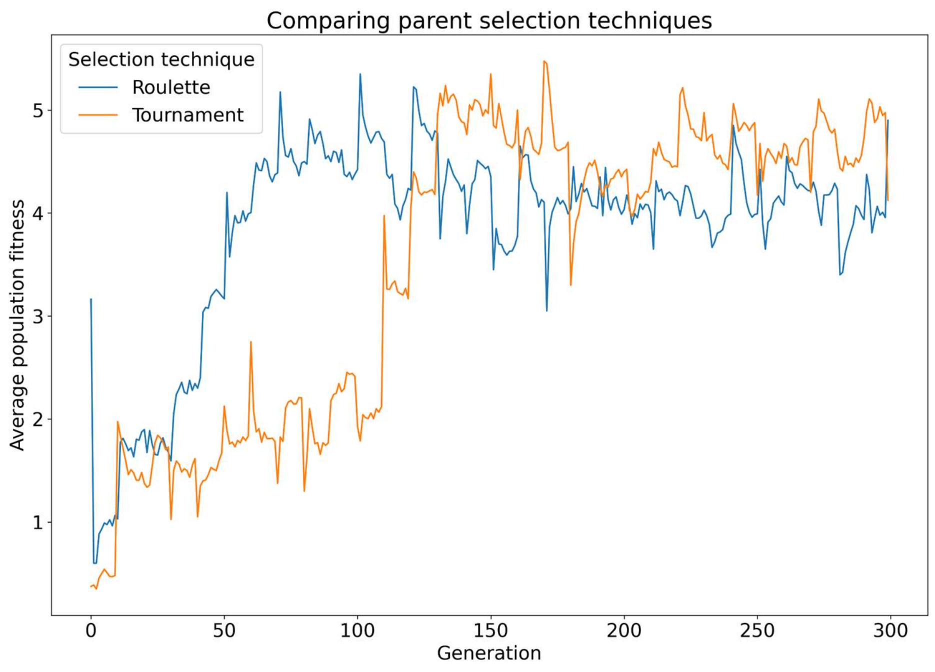


Figure: Comparing average population of roulette and tournament selection over generation

While population bred with tournament selection appears to take longer to mature to a good average fitness than roulette. Tournament selection does seem to have a slight edge over roulette selection in the long run, which is why I chose it over roulette selection.

Results and Conclusions

Performance

For clarity, the graph below shows the average fitness of the model using k-point crossover and tournament parent selection. I added a sigmoidal trend curve to illustrate how the model's performance increases initially before fluctuating around a relatively stable mean value.

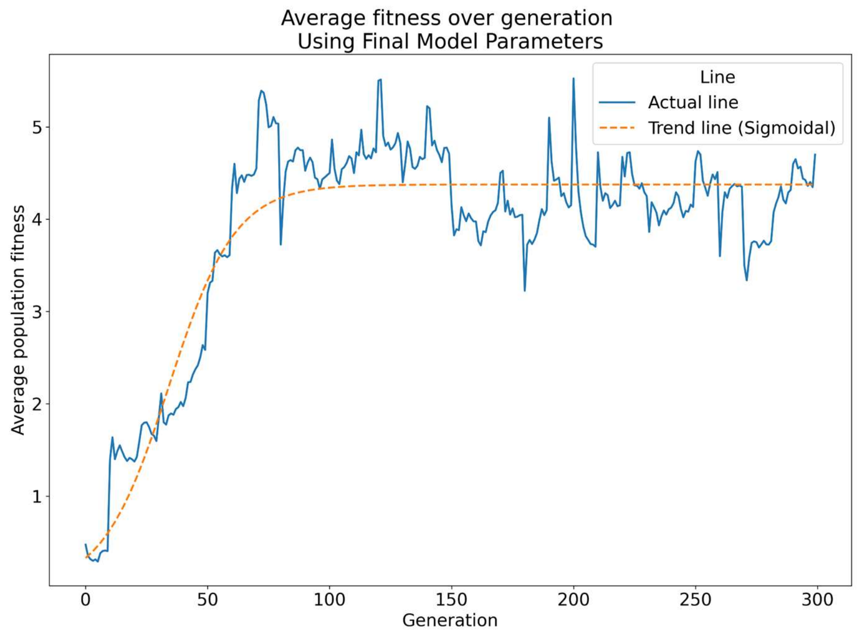


Figure: Average population using k-point crossover and tournament selection over generation. Trend line added for clarity.

Behaviour

When running a game against an opponent population, cleaners tend to (to put it lightly) exhibit far from optimal behaviour. Instead of exploring new squares, they simple travel back and forth along straight lines – never moving off the line they started at.

One could speculate that this is due to the simple model that I adopted (being pretty much linear with respect to weights and percepts). For the cleaners to exhibit more nuanced behaviour, it may be necessary to use multiple layers, using a variety of perceptron activation functions (e.g., sigmoidal, sinusoidal, exponential etc).

Additionally, this may be up to the fitness function. For example, one could choose a fitness function that rewards exploration and total number unique tiles visited (making sure of course to recharge!).