Jakub Skalski

Mars Lander

14th November 2021

Introduction

This report we will focus on applying a genetic algorithm to the *Mars Lander* game on the CodingGames platform. All mentioned algorithms are implemented in Javascript. The custom built game engine is able to handle 300000 simulations per second (measured with CodingGames during the preprocessing phase). We will be using a population of size 500 and the horizon of length 200 (or 100 if we think of a gene as a pair of angle and thrust).

Genetics

We start off by initializing the population with uniformly spread genotypes that would supposedly cover the most of the search space. We proceed by selecting the most promising subset of said population, then performing crossovers over its genotypes and finally mutating thus obtained genotypes.

Selection

Selection is performed by simply selecting $\sqrt{n}$ best individuals, where n is the size of the population. This differs a bit from the usual solutions where we would pick half the population and perform crossovers producing double offspring from a single pair but in general the only reason for this is to try all possible parent combinations. In the end this approach yielded the best results. Roulette was also tested but it didn't seem to improve the algorithm overall.

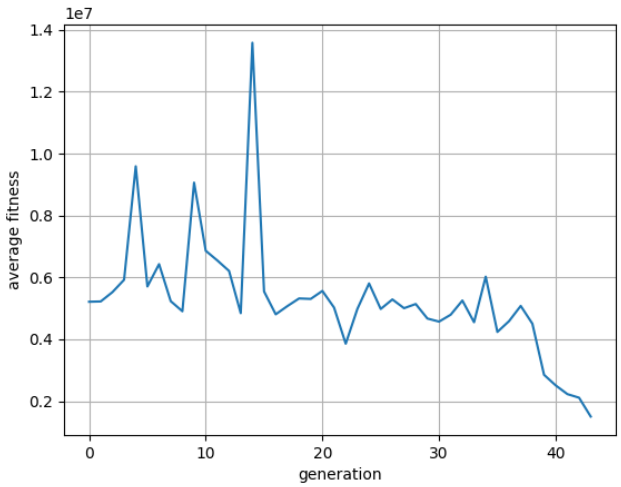
Crossover

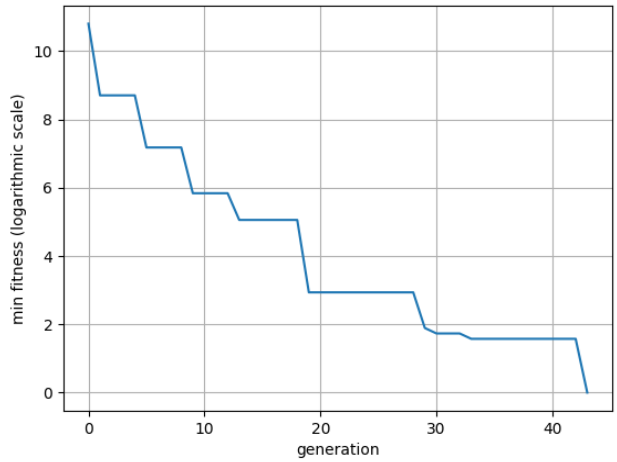
From the selected subset we take each pair and perform continuous crossover described in greater detail in A Dedicated Landing with a Genetic Algorithm. As mentioned earlier each crossover produces only a single offspring and should that offspring be created between a pair of the same genotypes (identity case) the offspring will not be mutated and is copied directly to the next generation instead.

Mutation

As it turns out, preserving a high degree of variance across the population is crucial, in other words we want the mutations to be impactful and introduce variety to the population. The most naive solution of swapping just a simple gene simply wasn't impactful enough. Also swapping multiple genes at random with uniformly distributed values also failed to work since we would usually get (in a sense) very similar genotypes. For example picking a random angle from range [-15, 15] rarely produces an interesting genotype, since the angle adjustments would usually mostly cancel out resulting in a wobbly but directed trajectory. For the purpose of solving this problem the mutated gene always sets the thrust to maximum and the angle is always either -15 or +15. We then spread this mutation across a few consecutive genes so that the overall effect is preserved.

Evolution progress visualizations





Fitness function

Intuitively such a function would make use of angle, speed and the distance from landing point since they are the values we want to optimise, thus the most naive approach would be to just sum them up. We can already see some problems with this solution since it doesn't take into account the range of values for each of these parameters, for instance distance would completely dominate the formula since its values range from around o to 10000 while angle is only from -90 to 90 etc. Obviously we need to introduce some coefficients that would balance the values. There are still some problems though, let's say a given genotype is doing pretty well in the angle department then it would probably be better to focus on the other values and adjust the angle when the other values have caught up. It would seem some sort of weight shifting is needed. The used approach is to take squared error of the values we are interested in, which succeeds in encouraging the algorithm to evolve in the more desirable direction. Some additional tweaks like turning off optimization for the values that already met set requirements and returning negative value equal to the leftover fuel in case the solution is found (so that we can try to optimise fuel next).

```
function MSE(angle_coeff = 100, speed_coeff = 10) {

	const
	sq = (x : number) => pow(x, 2),
	ang = abs(get(state, angle) as number),
	[sx, sy] = landing_spot,
	[x, y] = get(state, pos) as vector,
	[hs, vs] = get(state, speed) as vector,

	dist = sq(max(abs(x - sx) - 500, 0)) + sq(y - sy),
	bhs = max(abs(hs) - 20, 0),
	bvs = max(abs(vs) - 40, 0)

	return dist + angle_coeff * sq(ang) + speed_coeff * (sq(bhs) + sq(bvs))
}
```

So.. how good is it?

The algorithm passes all tests and validators and scores 255th place with 2214 leftover fuel: