# HappyBirds

**Introduction**

Happybirds is a game where you control a slingshot shooting birds at floating boxes midair. It's basically an Angrybirds clone. For the sake of simplicity here are some of the underlying rules that both player and AI must follow, as well as some facts about the game setup.

- SlingShot has 3 birds
- Only one bird can be flying at a time.
- When a bird goes offscreen(exception is the upper side of the screen, they can go offscreen there) the bird dies.
- There is no bouncing collision between birds and boxes, birds follow gravity and speed, boxes disappear on impact.

**Problem**

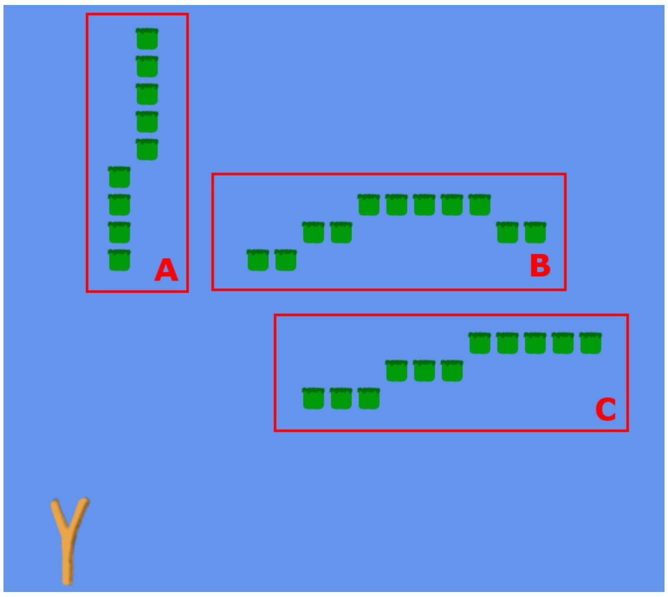
The default map consists of three sections of boxes (See picture 1). A typical good way a player would shoot their three birds could look like this (See picture 2). The problem here is to find a good way to remove all or as many of the boxes as possible, in only three throws. A throw has three underlying variables.

- Power (0 to 50)
- XDirection (-1 to 1)
- YDirection (-1 to 1)

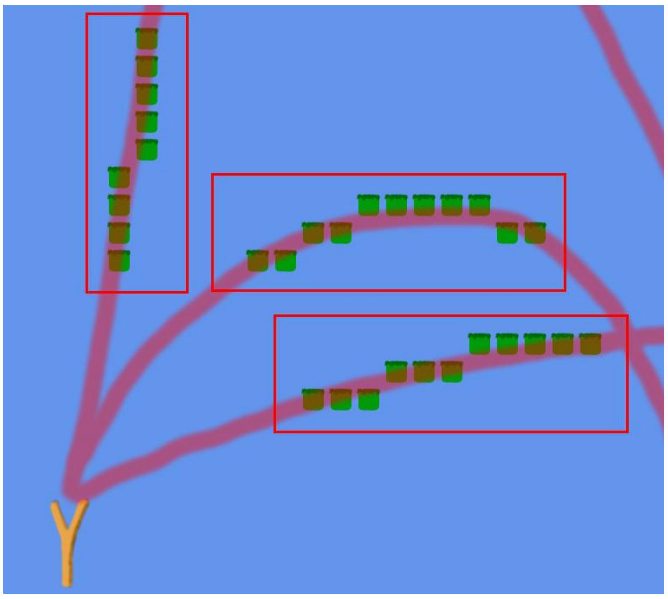
**Controls**

The player indirectly set these variables using his mouse, the further away from the middle of the slingshot the bird is positioned the higher the power variable gets, up to the cap. The directional vector between the mouse and the middle of the slingshot, inverted becomes the XDirection and YDirection values for that shot.

The GA works within the same scope of values and always plays within the cap. However the GA doesn't use the mouse, and instead directly set and change/mutate the values of the three inputs.



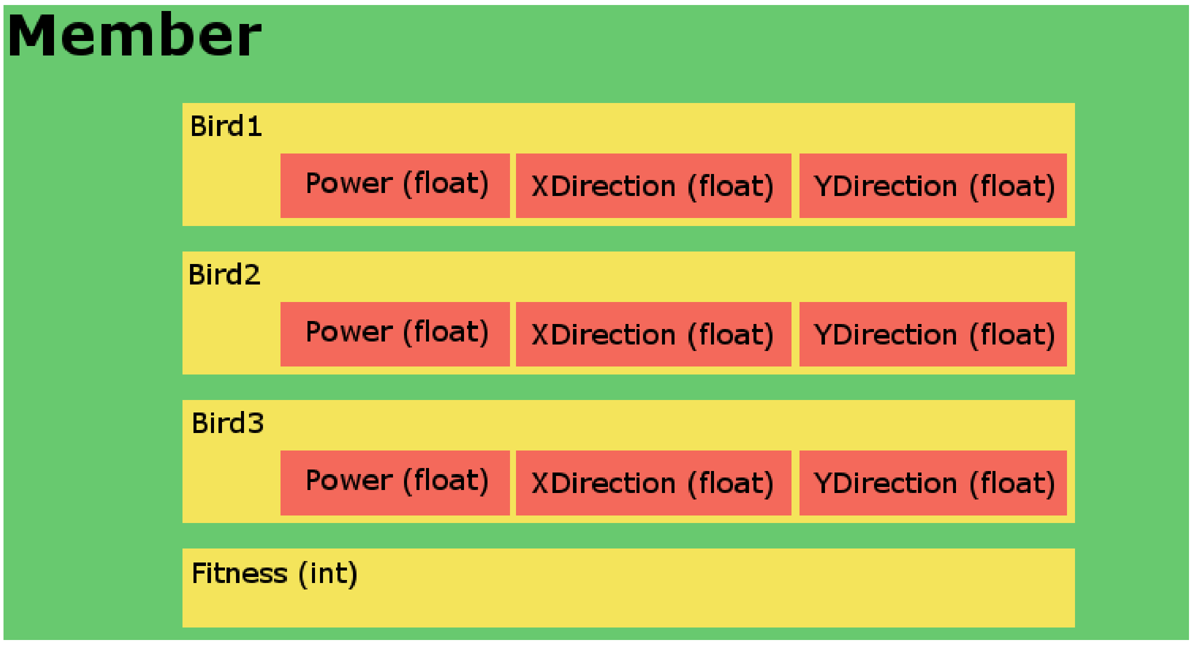
(Picture 1.)



(Picture 2.)

**Representation**

The member class stores some values to successfully be able to make three throws with different variables on each one. (See picture 3). As mentioned before the values are capped to a min-maximum of 0 to 50 for power, and -1 to 1 on XDirection and YDirection. The fitness is also stored in the member class.



(Picture 3.)

**Fitness Function**

The fitness function is very basic, after all three throws the GA will look at the level object and get the amount of missing boxes. So if a member manages to take out Section A, which contains of 9 boxes, that member will get the fitness of 9, after finishing the last throw. For the sake of simplicity I kept the fitness function simple. Things that could have been added to further enhance the calculations could have been:

- Gain fitness with time spent per throw (Would make GA try to extend each throw to not go offscreen for as long as possible).
- Gain fitness per spared bird. (In a scenario with less boxes or possibly more birds to throw, using less birds to remove all the boxes is a good thing, should give fitness).

With these in mind I wouldn't store the fitness as a integer in my member class anymore.

**Evolutionary Parameters**

When a generation is done it will go into a method called *AgeGeneration*, which consists of different types of procedures to mix up, mutate and flourish the different members of the member pool.

**1. Sorting**
We start with sorting all the members to easily manage top and bottom fitness values. Because we want to keep high fitness members.
**2. BreedTopGeneration**
We breed the top members by copying the upper half of the list and overwriting the lower part of the list with the copy. We now have 2 of each of what used to be 50% best fitness members of the original collection.
**3. MutateZeros**
This function speeds up findings of fitness by heavy mutation, close to fully randomness **ONLY** on members of the previous generation that turned up with zero in fitness. It's important that this step is made before 4. and 5. because of the fitness is reset to 0 after those functions are done.
**4. CrossOverHalfGeneration**
Now the breed half will be Crossover edited. Crossover happens on throw-level, two members crossover-ing will choose a random throw 1-3 and swap them.
**5. MutateHalfGeneration**
Now the breed half will be mutated at a random location in the member (all 9 fields can be mutated See Orange boxes Picture 3). Mutation works differently depending on which variable is mutated. Common for all mutations however is that it keeps the old value and tweaks around it, it's not a new randomized number.
**6. TwentyPercentRandom**
Now the breed half will have a percentage of its members fully randomized (20% of the total members, So for 20 members, 10 will be newly breed, 4 of the lowest fitness members will be fully randomized.)

**Results**

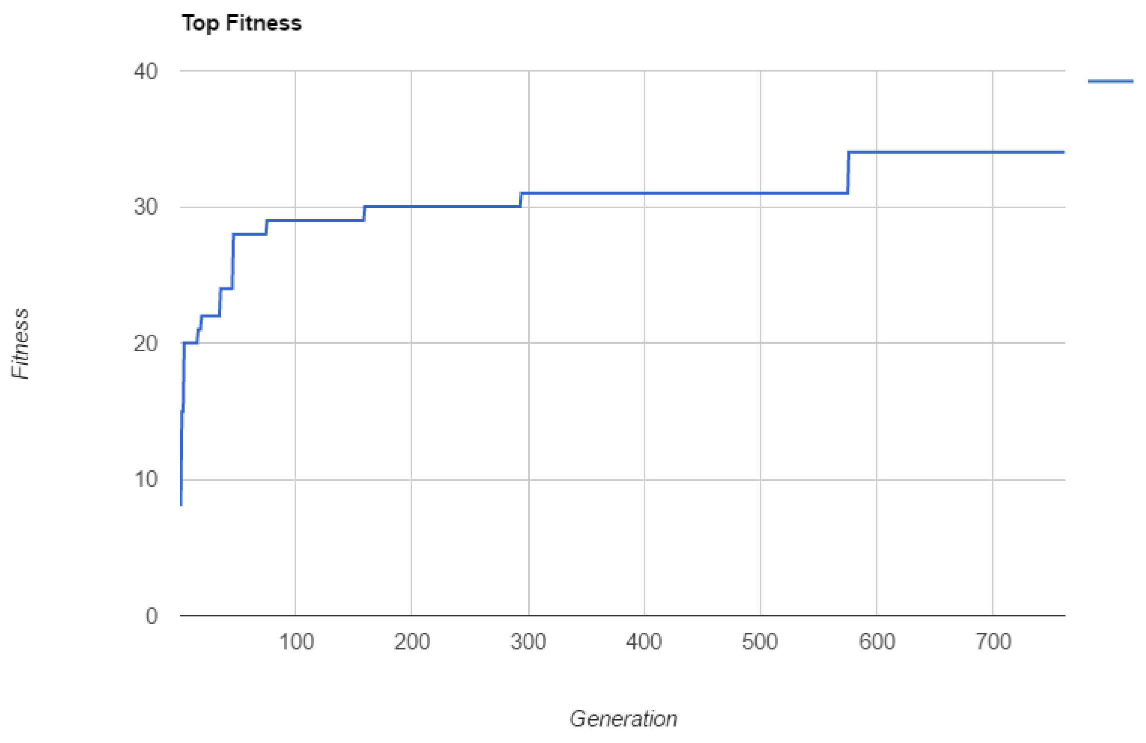
Due to a member being able to take up to 24 seconds to make 3 throws (if it throws them straight up) and the average time for a member to take their throws clocks in at about 5 seconds. 1000 generation takes 27hours~ of runtime. I stopped at 750 generations.

| Generation/Type | Top Mem. Fit. | Average Mem. Fit. | Median Mem. Fit. | Lowest Mem. Fit |
|---|---|---|---|---|
| 1 | 8 | 1,25 | 0 | 0 |
| 10 | 20 | 11,0 | 13 | 0 |
| 100 | 29 | 16,25 | 22 | 1 |
| 750 | 34 | 18,4 | 28 | 0 |

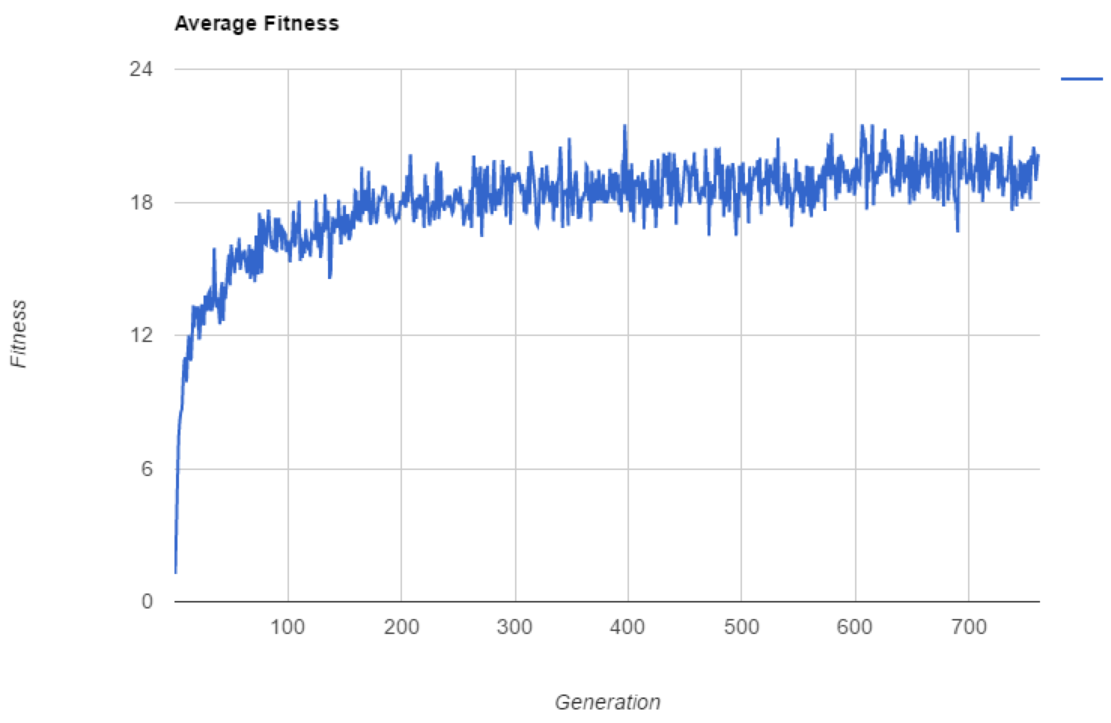
Mutation and Crossover makes the average spike back and forth when a generations new breed failure to maintain the fitness with their new changeup. The lowest varies very randomly because of my *TwentyPercentRandom* function. In the beginning the generations lowest member will almost always be zero. We can see that the first 20 generations had lowest fitness of zero. After that it becomes less and less common up to a certain capped percentage which is the randomness of my *TwentyPercentRandom*, thus the spikiness.

At generation 575 the fitness of 34 was achieved, that is one box away from scoring all of them. (35 boxes). My guess is that it would be possible to find the optimal solution within 10000 generations, however it's also shown how the progress is spanning out, slowing down.

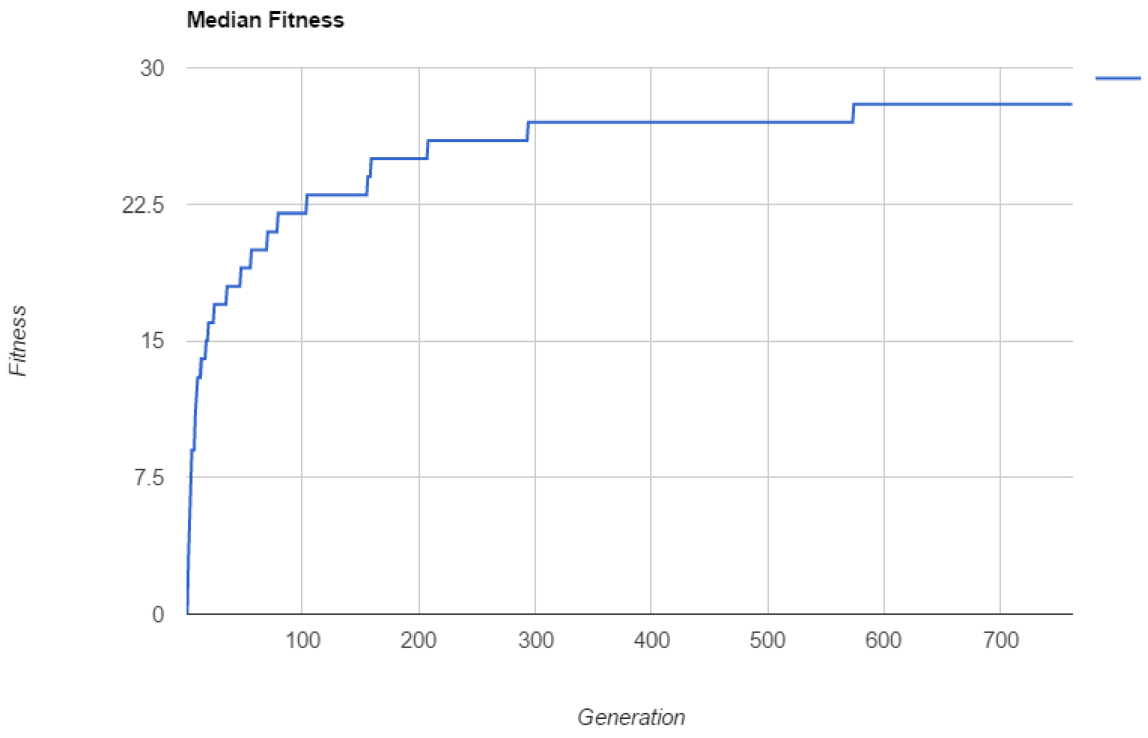
**Top Graph**



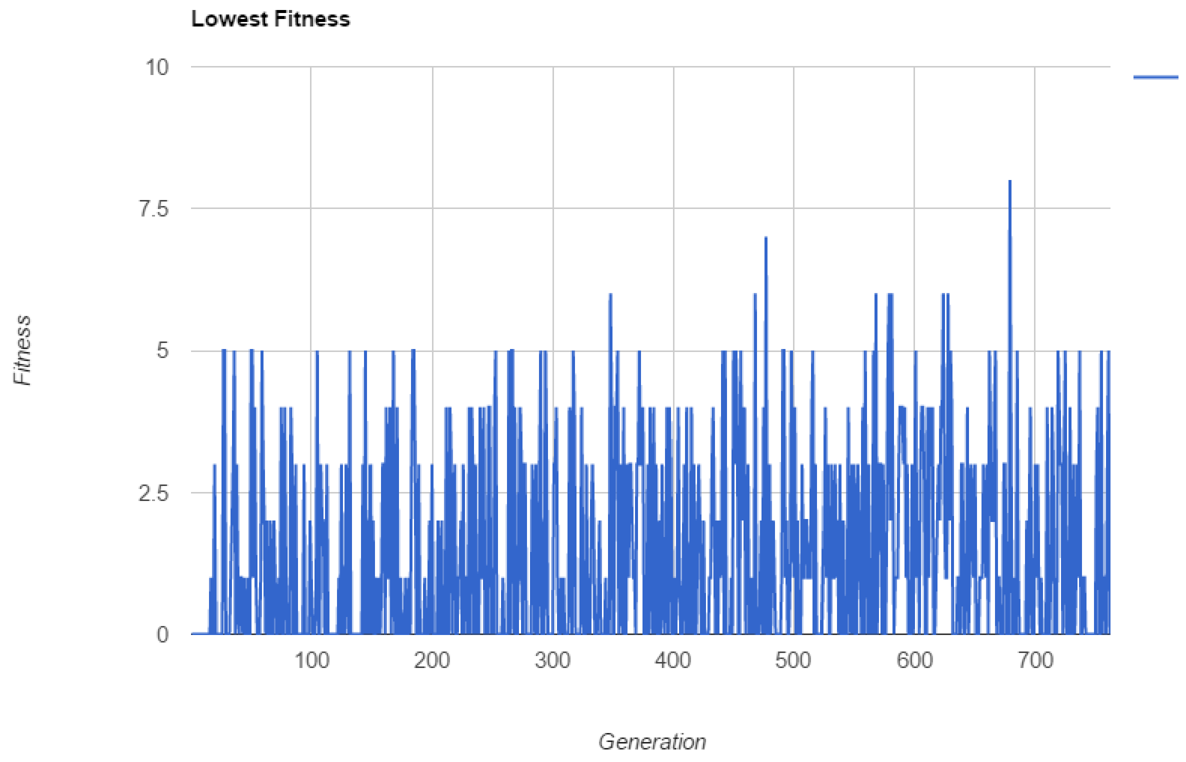
## Average Graph



## Median Graph



**Lowest Graph**


Lowest Fitness
Fitness
Generation