Exploring Genetic Algorithms to Solve Simple Games

Team Snek:

Abhishek Hanchate, Rylan Hunter, George Mikhaeil, Austin White

Texas A&M University

I. INTRODUCTION

A. Genetic Algorithm

A genetic algorithm is a metaheuristic inspired by Charles Darwin's theory of natural selection. It belongs to a larger class of evolutionary algorithms. The fittest individuals are selected for reproduction in order to produce offspring of the next generation. The process of natural selection starts with the selection of the fittest individuals from a population which then produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than them and have a better chance of survival. This process of selection based on fitness keeps on iterating and at the end, a generation with the most adapt or fittest individuals will be found. A similar notion is applicable to search algorithms where we consider a set of solutions and try to find out the best ones out of them.

B. Benefits of Genetic Algorithm

Genetic Algorithms are mostly used in situations where one can rely on generating high-quality suboptimal solutions to optimization and search problems.

C. Pygame

Originally written by Pete Shinners, Pygame is a cross-platform project of several Python modules designed for building video games. It consists of computer graphics and sound libraries which are made compatible to run with the Python programming language.

II. SMART ROCKETS

A. Game description

Smart rockets is a game wherein "rockets" (represented as circles) are generated and placed at a specified starting location. A specified "goal" location is placed somewhere on the map of available spaces. The desired outcome is for the rockets to learn the path towards the goal, while getting around obstacles that block the path. If the rockets hit the edges of the map, or run into obstacles, they will be considered dead and stop moving.

B. Why use a Genetic Algorithm for the problem?

Genetic Algorithms work well for this type of application, where we don't necessarily know the best path to take, but we have a way to score each individual's "fitness" to determine how close it is to the goal, and how quickly it made it there.

The most fit individuals are chosen to be parents in the next generation, where the parent parameters are randomly selected and mutated in the children. Assuming our fitness scoring algorithm and mutation algorithms are good, this allows us to improve our AI as we iterate through more generations, without ever having to actually know the best path to take ahead of time.

C. Parameter descriptions

We assign each individual rocket a position, velocity and acceleration vector, and then keep track of if the rocket died, whether or not it reached the goal, how many steps it took in the round, its final distance from the goal, and what its calculated fitness score is.

Our fitness score algorithm scores differently depending on whether or not the rocket actually reached the goal. If it did not reach the goal, then the score generated is inversely proportional to the squared distance from the goal (so the closer to the goal, the higher the score). If the rocket did reach the goal, then it's score is related to how many steps it took to reach the goal (with fewer steps leading to a higher score). These different scoring methods allow us to pick parents that will lead us to having better performing children in the next generation, even when we don't yet have a rocket that makes it to the goal. The program also terminates once a rocket has reached the goal and begins to calculate related fitnesses. This is because if a rocket makes it, no-matter how inefficiently, it automatically is the most efficient dot of the population, being the first to achieve the goal. Below is our fitness function for each case.

In case when goal is reached:

$$fitness = \frac{1}{16} + \frac{10000}{(Number\ of\ Steps)^2}$$

In case when goal is not reached:

$$fitness = \frac{1}{(Distance\ to\ Goal)^2}$$

Each rocket has its own unique "DNA," which is a list of vectors for the rockets to follow. These vectors are essentially applying forces to the individual rockets, moving them through space. Initially each rockets DNA is completely random, and then we score each individual rocket after the round using the fitness algorithm discussed above. We then select the best rockets to be parents to the next generation, where we take traits from the parents and randomly select parent traits to vary for each child. We then iterate through the process with the children and choose the most fit children to be parents to the next generation. This process is repeated until we find an optimized model where we are no longer improving from generation to generation.

D. Playing around with the parameters

We found that changing the parameters (such as population size, obstacle amount/location, and the fitness scoring algorithm) affected how quickly an acceptable AI was found. A smaller population for example means that there are fewer mutations in each generation, so it may take significantly longer for a successful AI to be found (in the case of a single obstacle game, a population of 100 took over 350 generations to find an AI as successful as generation 30 when a population of 1000 was chosen, while a population of 5000 was equally successful in only 15 generations). We also found that the more obstacles we added, the longer it took to find an acceptable AI. However, this was expected, and is actually where genetic algorithms are more useful, since it would be significantly harder to program an AI to get through the more difficult course.

E. Single Obstacle Game

We first implemented our algorithm on a simple course with a single obstacle to navigate. The goal is for most of the white dots (smart rockets) to reach the red dot (target).

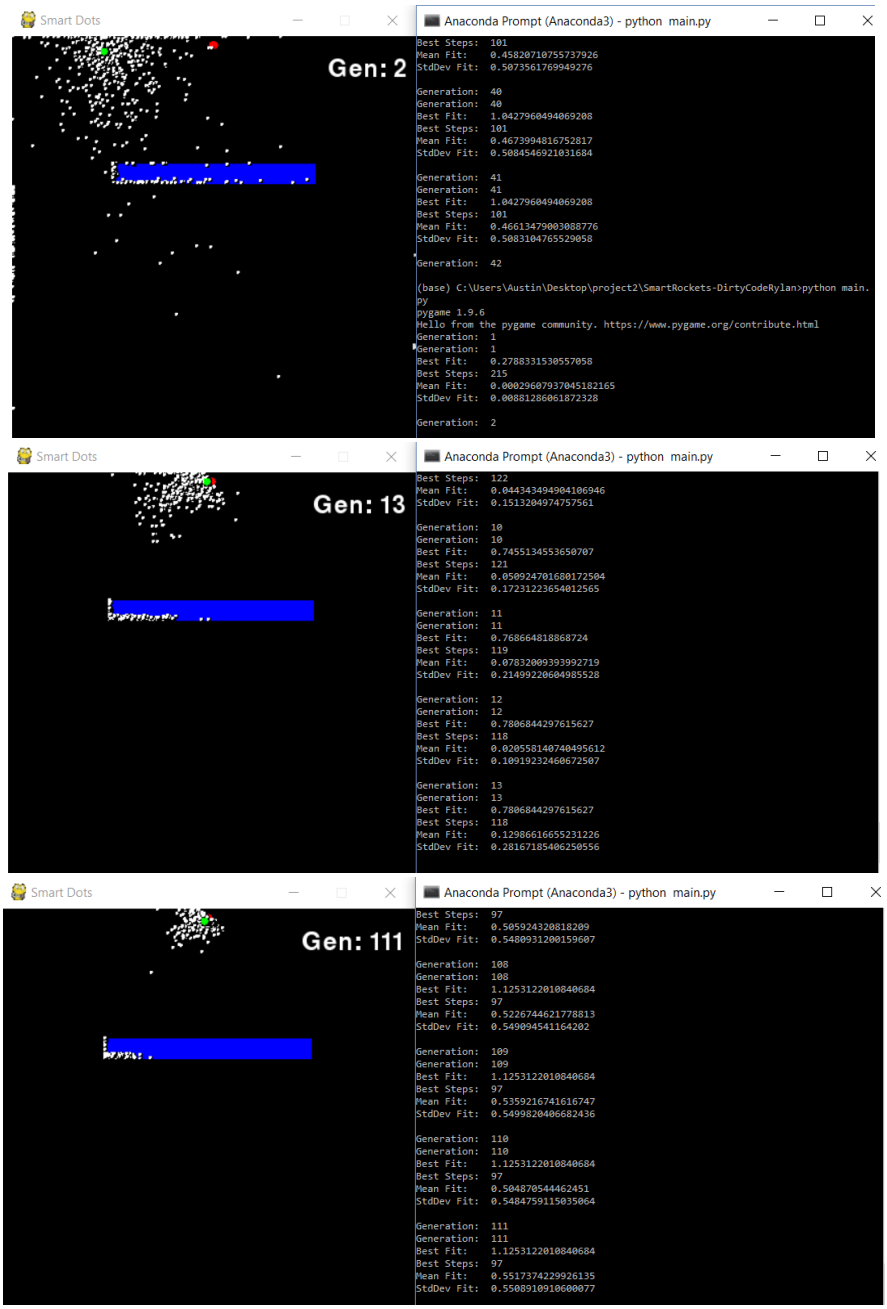

Fig. Single Obstacle Smart Rockets Game

The above images show what the game looks like with a single obstacle, as well as the terminal outputs for each generation's round. As can be seen, initially the rockets did not fair very well. But since it's a simple obstacle, a rocket found the goal in only the 2nd generation (although it was very inefficient). By around the 30th generation the model began to plateau in efficiency, but I ran it for 111 generations to demonstrate it can continue to slightly improve for many generations, but it does get to where the improvements are negligible for the application.

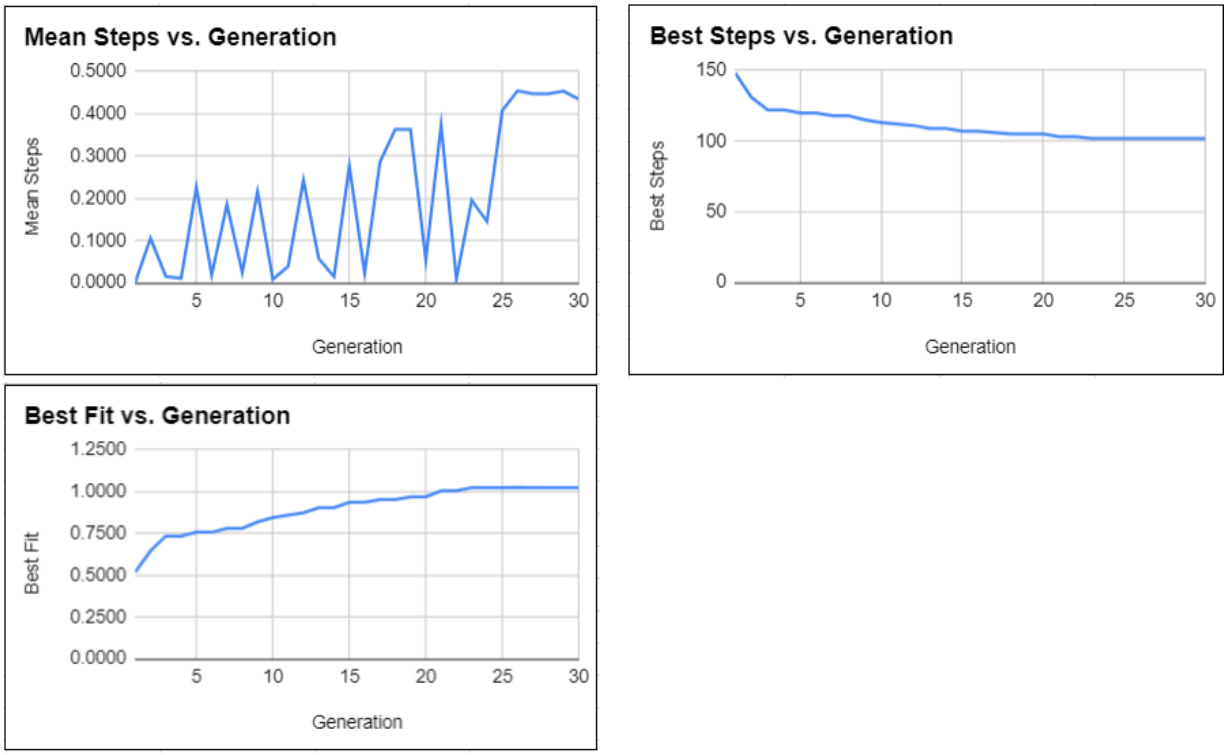

Fig. Best Fit, Best Steps, and Mean Steps vs Generation Plots

The graphs above show how the most fit model of each generation improves over the first 30 generations. It can be seen that sometimes there is no improvement from one generation to another, and then there will be improvement still in later generations. This is due to the DNA randomization, sometimes there won't be any better models in the next generation, so the previous best model stays as a parent for the next generation until an individual scores higher.

F. Multiple Obstacle Game

We then tried our algorithm on a more complicated course, as genetic algorithms are more useful when the optimal path is more difficult to program.

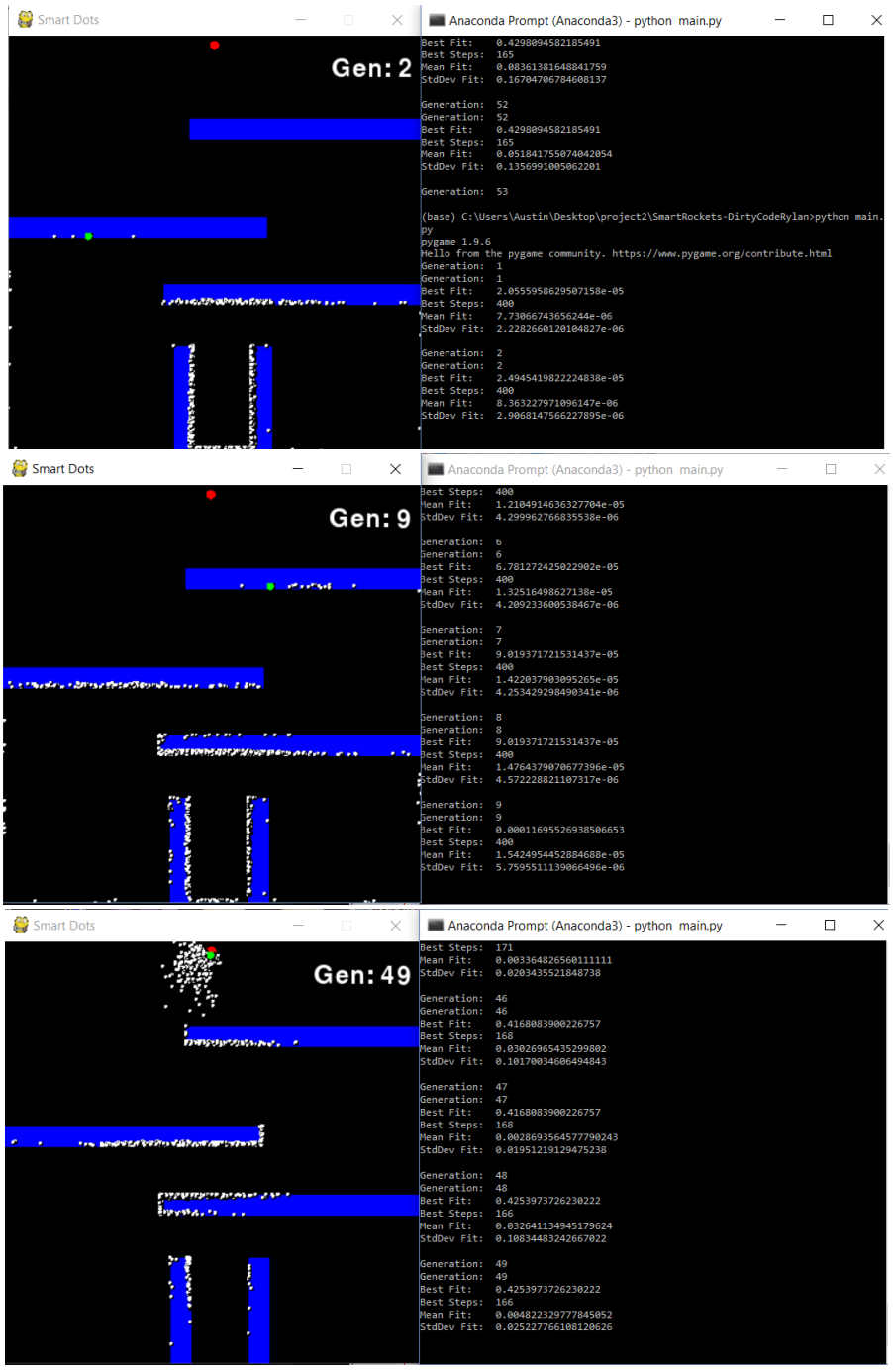

Fig. Multiple Obstacle Smart Rocket Game

The images above show the algorithms progress in generation 2, 9, and 49. As can be seen, it took significantly more generations to get the rockets to efficiently reach the goal. However, even when every rocket died early on in the course initially, a path was able to be found that could finish the course. Initially the rockets that made it to the goal were very inefficient, taking needless turns often. Due to the way our scoring algorithm works, we were able to improve the efficiency fairly rapidly through further generations.

G. Limitations

As mentioned previously, it is helpful to have a large population in each generation, as this makes finding an efficient AI take few generations. The computer's available RAM limits how many can be used, which limits how quickly the AI can be found. This game is simple, so a large population is easily attainable, but on more complex games with more memory requirements, this could be an issue. I found I could get a population of about 4000 on my computer without significant lag or crashing occurring. Another limitation is how the fitness is calculated. If you use a poor fitness function, then this method may be very ineffective at finding an acceptable AI.

H. Conclusion

Our genetic algorithm was very successful in finding an efficient path to the goal, and even with more complex obstacles in the way we were still able to get an AI that completed the course efficiently. By adjusting our fitness function, as well as our method of parent selection and trait randomization in the child generation, we found a model that fit our needs and was also efficient in how many generations it took to find an acceptable AI.

III. REFERENCES

Smart Rockets and Genetic Algorithm tutorial:
Code Bullet, "How AIs learn part 2 || Coded example," *YouTube* Available: https://www.youtube.com/watch?v=BOZfhUcNiqk.

IV. APPENDIX

Our Code Repository:
https://github.com/InformationScience/FinalProject-teamSnek