Daniel Glass

Doctor Mazidi

CS 3340.003

14 April 2018

Genetic Pathfinding Project Report

It may be necessary to preface this report with some disclaimers. This project is intended to belong to a class of algorithms called “Evolutionary Algorithms” (EA). These algorithms consist of a few key steps: reproduction, mutation, recombination, and selection. Although this program contains these steps, due to unfamiliarity with EAs, this term is very loosely applied to this project. This project is merely an effort to illustrate an EA in a simple environment.

This project simulates a species whose sole purpose is to navigate through terrain to get as close as possible to a food source. Initially knowing nothing, the species will move around randomly. After the moves are complete, the species’ fittest members will reintroduce their genes into the next generation through methods of both recombination and cloning. The next generation’s genes will then be mutated by a certain amount. The next generation goes through the same process with a higher count of, hopefully, fitter individuals. This process continues, in this case, until the user terminates the program.

The program begins by initializing the terrain. This consists of building a border around the traversable space (a 64 by 64 grid), then randomly placing a modifiable number of walls as obstacles around the inside of the traversable space. This terrain generation is to add some complexity to the program, and to provide an actual problem to overcome. One could eliminate a majority of the problem solving by drastically decreasing the wall count. This would provide quicker learn time, as there are no walls to inhibit a direct approach. The intended nature of the project, however, was for the population to navigate around obstacles. Due to this, the genes of the species were designed for complex movement. For a species without a need for complex navigation, the gene could simply consist of a direction.

After terrain initialization, the species gene pool is initialized to a sequence of 4096 randomized “directions”. This is for 128 movements for 32 animals. These directions are simply the numbers 0-3 corresponding to each cardinal direction: up, down, left, right. For this, two bits are all that’s necessary. Since most of the operations were reading, writing, and modifying the gene pool, however, simplicity of the genes was a must. As such, a design choice was made to make each movement a byte. If one was dealing with a large population or a large gene sequence, efficiency would’ve been priority, but readability and simplicity was priority in this case. Overall, the space taken by these data elements holds 5248 bytes compared to 1312 bytes if it was optimized. Memory being as cheap as it is and the small size of the simulation, a 0.003936-megabyte memory difference didn’t seem to be too big of a concern. This also has some affect on processing time, but a delay was put in place so that the user can see what happens throughout each generation. For that reason, speed wasn’t a huge concern either.

After the initialization of the gene pool, the user inputs the desired x and y coordinate for the food. This is then drawn on the map. Next, each animal has its x and y coordinate set to the center of the screen (31,31). The coordinates of the entities on the screen (the animals and the food) were stored as two separate bytes. In the case of the food, only one x and y pair was necessary. For the animals, a list of 32 bytes (for each animal) was used for each axis for a total of 64 bytes for all animal positions. A byte was used for the position because a number greater than 63 was needed for the coordinates of an entity. A half-byte wouldn’t have been enough. Again, only 5 bits were necessary for the coordinates. Another design choice was made for the reasons aforementioned.

Following the drawing of the animals’ initial positions, the simulation really begins. Using a nested loop, each animal has each of its genes run through. If the desired movement is blocked by the terrain, that move goes to waste and the next one is read. For this, a simple collision check is used by comparing the color of the next block to the color of a wall. Each of the animals’ 128 movements are run through and the trail of the animals’ paths are visible. Each time the animals succeed in a movement, their corresponding x and y coordinates are saved back to the animal\_X and animal\_Y list. After each gene for every animal, the program pauses for 5 milliseconds to let the user see the path they take.

After every animal has completed its movement, their final positions are used to calculate fitness. For calculating fitness, a simple linear distance was used. This could possibly lead to the animals being stuck behind a wall, not being able to mutate out of their final position. This was another design choice made for the sake of simplicity. One could use a pathfinding algorithm and calculate the distance of the path from the animal to the food and use that for fitness. This would prevent them taking any bad routes, but it was a bit too hands on, as this program was intended to learn with minimal interaction. Also, implementing a pathfinding algorithm for each animal would use a lot more processing power than a linear distance calculation. After calculating linear distance, the animals that were closest (less linear distance) were defined as fitter. The animal coordinates were looped through to find a maximum of 4 of some of the closest animals. This is done by storing any number less than its preceding number up to 4 numbers. The method used to find the least distance doesn’t go through the population more than once, so the true best 4 won’t be found in some cases, but it works well enough.

Once the top animals are found their genes are stored into a space of 512 bytes to hold 4 animals’ genes. These genes are then copied over to a variable to store the previous generations genes because the original will be used for splicing. In splicing, the 1st and 3rd are spliced, and the 2nd and 4th are spliced. The splicing process includes finding a random position to splice, going to that position in both genes. After that, the front of one is combined with the back of another and the two resulting animals are considered their offspring. Initially the genes passed on would only be those 4 offspring, however some modifications had to be made for various reasons. This will be explained more in the following paragraph. The next generation includes the following: the previous generations fittest, the offspring of the fittest, clones of the offspring. The population is initialized like the first time. After it is initialized, the first 12 animals (the first 1526 bytes of the next population) replace the random first 12 from the next population. Finally, the entire gene pool, minus the previous generation, is mutated. The population then simulates over the last one and the steps are repeated.

Many things about this program would be changed it were made a second time. Readability could’ve been greatly improved by building many more functions. Some jump statements should’ve been functions, but using a stack pointer seemed a bit unnecessary, albeit easy to implement, for a project of this size. Some of the methods used in this program could’ve been more generalized in a way that many parts of the program could be run through one function. Some design changes would be made which as well were avoided for simplicity’s sake. Adding multiple food objects and allowing many different breeds of animals to emerge going for different food sources would be neat. The method for selection of the next population would be changed as well. The selection of the population used was built around the initial selection of 4 fit individuals. This method left holes, however, as an ascending order list of distances would leave only one animal as the fittest. This leads to some offspring with no genes, which will ripple through future generations. This is, some animals in the population will go straight up as an indication that their genes are all zeros. The preferred method would’ve been a function that took arguments such as the number of animals to be fit and maybe a fitness threshold. It would then do multiple binary searches for the least distance however many times specified as it stored them to variables. The distance calculated for fitness would also be changed to the distance of a found path from the animal to the food. This would prevent the animals from getting stuck like you may observe in some situations. The next generation’s number of fit individuals was initially 4, but that lead to far too many random animals doing nothing for the simulation. It was changed so that the parents were carried to the next population (this is so that progress would not be forgotten by any random mutation) and the amount of offspring was doubled (this was so more of the next population was making progress). A change for the end case would also be made. Currently the user needs to end the program to stop the simulation. A preferred method would be to compare the fittest from the last population to the fittest of the current population. If they were identical, the simulation would be assumed to have stagnated and no more growth would occur. The program, initially, wasn’t desired to end when the animals got to food because the gene sequence was 64 and the most optimal route was desired for the animals. This, however, lead to an unnecessarily long simulation time. This is due to the need for a near perfect path to get to the food source if it was in the corner. This didn’t illustrate the concept well, so the number of moves allotted per animal was increased. Since the animals now make it to the food, an indication of a “successful” animal would also be nice.

Some design changes would be made if aesthetic was a major concern. A method for a cleaner representation of past populations would’ve been a good touch as well. The method used was the best of a few. The population could’ve not shown trails and only show the final positions of the animal in that population. It also could’ve shown the animal’s movements through the simulation as if they were actual animals moving around. This was the preferred method, however due to the processing time between the initial animal placement and the updated animal placement, a temporary animal coordinate list would need to be used to be able to place the next movement and erase the movement prior. This seemed a bit excessive for a visual improvement, and this method still didn’t illustrate, well enough, the previous generations and the progress of the simulation. The colors of the simulation were chosen to emulate a grassy field with rocks, but that can easily be changed by making the hex codes assigned to colors different. My preferred color scheme (black walls and a white background) is what it is for visual clarity.

A few of the methods used for display and translation from coordinate to an address was inspired from another project in MIPS. The program was a clone of the old Snake game. The program tested collision by the next pixel’s color, which this program’s collision check was inspired by. The program also used a generic method of setting pixel color and change from a coordinate to an address, and it was done very well. Due to those reasons, it was found acceptable to use those as inspiration seeing as they are the best way of going about things. The program also used things like setting the screen to black and clearing the registers that are simply remnants of trying to see how the bitmap works. Over all, this accomplished 3 things the way the Snake program did when it came to the display of data. This program cites the Snake program at the top, however it’ll be included here for clarity. ( Snake project by Shane Shafferman and Eric Deas <https://github.com/Misto423/Assembly-Snake> )

Over all, this program accomplishes what was intended. This program was intended to simply show what a genetic algorithm does and display it in a relatively easy to understand manner.