Colin Thackston

Laura Huber

Alex Hayes

For our project, we decided to create an evolutionary algorithm that would optimize a car that can traverse a 2-D map dotted with varying terrain. This problem is incredibly interesting because it us to track the progress it makes in real time. We implemented a genetic algorithm implementing crossover, mutation, and migration to solve the problem. A GA is well sorted to this problem because there are sixteen continuously distributed variables which creates hundreds of thousands of possibilities. If we had discrete variables, then generating random solutions would be fairly quick.

Optimizing cars is not a novel idea. Several other open source sites have algorithms on genetic cars including <http://gencar.co/> and <http://boxcar2d.com/>. These algorithms are quite sophisticated and their physics implementation far surpasses ours, however, the concept is the same. We actually designed our track in the same way these programs did. They used Javascript and not Python though.  The goal was to create a GA and not a complex physics implementation in Box2D. There was no reason to quite literally reinvent the wheel.

To solve this problem, we implemented a genetic algorithm in Python using the packages Box2D and pygame, tracking sixteen attributes and mutating cars with crossover to vary the population and improve the flock between the generations.

When the program begins, it spawns an initial population of twenty cars.  Multiple population sizes were tested, but this number was ultimately chosen to produce enough subjects for genetic diversity while also being small enough to complete in a reasonable amount of time.  Large populations produced an unhealthy starting state where most of the cars died immediately (the “Hunger Games Cornucopia”), as many as 95% might die as cars attempted to crawl over each other.  Small populations had the opposite problem: not having enough diversity to produce interesting results, combined with needing many more generations to get anywhere.  The population number that was ultimately chosen produced a healthy balance where results could be achieved in a reasonable amount of time, and enough changes could happen between generations. From literature, we knew we needed a population closer to a thousand, but there was no way we could implement such a large number.

We built of sixteen attributes into each car and randomly assigned their values to a car at the start of the program. We bounded all variables to prevent any cars that we knew would not works (wheel radius or density, chassis density and axis positioning, motor speed).  These constraints were chosen to produce results that would act like cars but have enough flexibility to breed toward a more optimal solution.  Though this initially sounds limiting within the scope of what a genetic algorithm can produce, these rules are necessary to achieve our ends.  Each generation is scored based on its ability to move a vehicle from point A to point B, but without limits the best vehicle turns out to be one that bypasses the hurdles entirely (a “Helicopter” that weighed nothing but had infinite wheel velocity and flew over the course).  Just as there are rules within sports or checks and balances within government, there need to be limits within our program to produce useful results.

Depending on the starting attributes, some cars will die off almost immediately while others crawl their way through the opening course.  The “death” of a car occurs when its health reaches zero, decrementing if the total linear velocity reaches less than 0.0001 (update\_health).  The total distance reached by each car is stored as a score, and when every car in a generation dies off, the cars are sorted by their score and the average distance is computed (sort\_by\_dist).  This serves a dual purpose: tracking improvements between generations and weeding out subjects that perform poorly.

Between generations, the top eight performing parents were retained to go again in the next generation, the next eight were children bred from the top 80% of previous population, then the last four cars were new randomly generated cars in order to work more diversity into the population.  Multiple methods were chosen and changed as we developed the final solution.  Initially the parents were bred completely randomly, but this could just as often merge the DNA of parents that performed extremely well with parents that sat on the starting line and died immediately, while also relying on mutation alone to supply diversity.  To correct for this, weights were put in place to favor subjects that performed well on the course (keep top eight parents), while subjects that barely moved were less likely to make it into the next generation (prune one the bottom 20% from breeding into the next generation), and then random cars being added into the mix helped diversity the population in a way that mutation alone could not.  Using these categories between generations ended up being the best way to improve generations over time, and could produce usable results without taking thousands of generations.

There are still some issues: a starting generation that performs well will trickle down into the subsequent generations easily, but a starting generation that performs poorly will often continue to perform poorly until random subjects begin performing well and passing their DNA along.

In general, the process of getting good, clean data proved to be a little harder than we expected. Our implementation of the physics package resulted in all of the cars starting at the same place on the GUI and some of them would start upside-down. Initially, we tried drastically scaling up the population size to five hundred. Our GUI could not handle that much data and the resulting lag would kill many of the better cars in the first few seconds. Increasing the population compounded the problem. To get more consistent data, we decided to decrease the population down to twenty which led to a higher percentage of cats getting off of the line. In a more traditional GA, we would have a population around a thousand which would presumably show a clean upward trend in population average distance. Even with a small population, only half could ever get off the line on good generation. This made measuring average population distance troublesome because the bottom half was always going to be zero. To fix this problem, we simply took the average of the top 25% of a population which produced a more reliable measure of the population. The figure 1.0 presents the population average distance of both the random and the GA.

As one would expect, the average distance of the randomly generated populations is fairly constant. Random will never improve on average. The highest average distance traveled by any random generation was 6.800272 which is not impressive. The GA achieved a highest average distance of 65.7540516853 which is 9.7x better.

The maximum distance traveled by a randomly generated car was 116.10597 compared to 147.533889771 in the GA. Not only did the GA create a higher average population distance (meaning there were more successful designs), but it also created a better car. It is clear from the graph of the GA data that not every generation was successful. As late as generation thirty, the average still occasionally plummeted to below 10.0. Once again, our physics implementation caused these results. We kept the top 20% from one generation to the next so one would expect a much smoother line of progress from one generation to another. Successful individuals did not have the chance to get off the line or started upside down which instantly killed them.

The initial generation bug proved to be a source of constant error and we spent a considerable amount of time trying to fix it. The most straight-forward way would have been to parallelize the process so each car ran in its own environment. Unfortunately, we lacked the hardware capabilities to fully implement the solution. Aggregating the data from over a hundred different processes also complicated the problem exponentially. We acknowledge that our implementation of the physics package greatly reduced the effectiveness of our end result, however, our GA itself proved to be highly efficient.

We initially implemented our GA using binary encoding of the cars. Using a function for converted the car object data into a binary string, we randomly swapped chromosomes one by one. This program executed quickly, however, it did not produce anything resembling optimization. The binary encoding actually produced results worse than the random sample and never converged anywhere close to the records set by the random or other GA. We believe this had to do with how we were swapping the bits in the binary string. Sometimes very significant bits would get turned off (or turned on) when they should not have been resulting in a child that looked dramatically different from either parent. Since only one “gene” controlled an entire attribute of the car, a small mutation or incorrect transfer could produce a massive and unpredictable result. In biological systems, multiple genes control important attributes. These built-in safeties stop small mutations from producing fatal results. Our encoding had no such strategy. A future implementation would encode the bits differently so that there were not as susceptible to mutation and to only mutate the least-significant bits. We were quite worried for a while that our GA was incapable of optimizing a population so we tested it by giving it a simple problem to solve: summing an array to a given result. Our algorithm quickly solved this problem regardless of array size so we determined the GA itself was not the source of any problems.

Overall, we are satisfied with our results. We built a GA that optimizes a car with a fitness measure directly applicable to the result we wanted. It created this car within a reasonable amount of time (~fifty generations) and we collected clean data about our results. We did not set a specific convergence rate because there was no guarantee that any car could complete the course. The course, after all, was randomly generated so within the given parameters we created it could have very well been impossible.

If we were to continue with this project, we could explore different areas of the problem. Our current fitness function only measures distance, but future iterations of this project could also determine fitness from other elements, such as time. In our tests, we found that many of the cars that evolved did indeed traverse a large amount of the map, but at a very slow rate. Including a second element in our calculation of fitness could potentially create generations that evolve very differently. A possible implementation of this could be the use of a fixed distance and the implementation of a timer to determine which builds completed the course in the least amount of time.

Another curious addition to the program could be the integration of three or more wheels to the vehicle, which would likely improve both speed and map maneuverability. Of course, the addition of these elements to an already complex system would prove difficult to manage.

A more difficult or randomized course would also be an interesting addition to our program. Currently, the genetic algorithm we use creates vehicles that become particularly adapted to the specific trials present on our one track, but the vehicles that do well on this one course may end up failing on a different or harder course. It would be interesting to discover how our GA evolves its vehicles in response to a variety of maps.

**Figure 1**