Introduction

For my undergraduate research project I set out to investigate the field of genetic programming, and build a simulator that could create joy programs. I decided to build my simulator in C#, and have it create programs in joy, a concatenative language. The majority of the semester was spent modifying and fixing the simulator. Getting an initial version up and running didn't take long, but there were many bugs and issues that prevented me from doing larger experiments with it. Once almost everything was fixed, I had a few weeks left to run the simulator on large populations and see what it could accomplish. At the end of this I'll go over some results from various experiments I ran.

Languages

There were two main considerations when choosing what language I was going to write my simulator in: speed and comfortability. Because of the nature of genetic programming and the amount of time it takes to produce meaningful results, I needed a language that would be able to quickly create, evolve, and score large populations of programs (>5000). If that was my only consideration, I probably would have chosen to write the simulator in C or C++, as those are both incredibly fast. However, because of the time constraints, I needed a language that I was already fairly comfortable with so that I didn't have to spend any time learning a new language. After some consideration, I chose to use C#, a fairly fast language that I also had some experience with from some other side projects I had done in Unity.

I then needed to choose what language the simulator was going to write its programs in. There were way more options for this than I initially thought, and I eventually settled on Joy, a concatenative stack based language. There were several reasons that I chose this language. Firstly, it being a concatenative language makes "breeding" or doing a crossover mutation between two programs extremely easy, as I can split the parent programs at any point and create a child from them. Because the language is stack-based, and any program can be written as a single string, it also makes it super simple to generate programs, as I can randomly put together any combination of keywords that I want. Joy's interpreter also supports file input, which allows me to write all of the programs generated to a file, and easily run them against all of the test cases and parse the output. Because of all these features, joy was a great choice for a first language to use for the simulator.

Program Flow

The overall program flow of the simulator is a fairly simple loop that continues indefinitely until a solution is reached by a program. Each simulator begins by creating a randomly generated population of programs. The majority of these programs won't even run due to various syntax and argument errors, but that's ok. Typically in the first generation about 90-95% of programs fail to run. Once the first generation has been created, they are all run with each of

the provided sample inputs in the test cases. Each of those results is then compared to the expected output provided in the test cases. The sum of the absolute value of the differences between the expected and actual output is then calculated as the score of the program. If the program returns an error for any of the input values, then it's given a score of -1.

Once all programs have been scored, they are sorted from lowest score to highest, and sent off to be evolved into a new population. During evolution, the top percentage of the population is saved and automatically put into the next generation. From there, the rest of the next generation is created through mutation, crossover, and deletion. That new population is then run and scored, and the cycle continues until a program receives a perfect score.

Challenges

Speed

The first major challenge I faced with the simulator was speed. In order to be able to solve meaningful problems, the simulator would need to run quickly with large populations (>1000) and lots of test cases (~50-75). My initial version was able to run 30 generations with a population of 100 in about 2 minutes. This was far from what I wanted. If I wanted to run thousands of generations with populations 50 to 100 times larger than that, I would need to make some serious optimizations. To tackle this problem, I looked into using a profiler to figure out where the bottlenecks in my code were.

After looking at the results, it was immediately clear what was holding the simulator back. Over 98% of the time the simulator spent running was spent on running each joy program. The majority of that 98% was spent doing file I/O and subprocess creation - creating a file, writing the program to that file, starting a subprocess to run that file in the joy interpreter, and reading the output from that process. For every single test case and every single program I was doing all of that once. That meant in my 30 generation run with a population of 100 and 15 test cases, I created a total of 45,000 files, and started just as many processes. This just wasn't sustainable for the amount I needed to scale up the simulator.

To fix it, I modified the entire running/scoring process of programs to do everything once per generation. This meant writing all of the programs formatted with the test case inputs to a single file, starting one sub-process to run that file in joy, and then reading and parsing the output from that process (This was a task in itself, more details below). Once that whole system was working, what used to take over 2 minutes to run took less than a second. It was exactly the performance boost that I needed to be able to solve larger problems.

Parsing Joy Output

As mentioned above, parsing joy output once I consolidated everything to a single file was a whole can of worms that I wasn't expecting. Before, if a program returned an error, I could disregard all of the output and score the program with -1. Now, since I had to separate and parse the output for each program, I couldn't disregard everything when I received an error. This

was an issue as errors produced by joy varied in size and length. Many were just one line, but some were up to three. This made parsing nearly impossible, as I assumed each program would produce one line of results. To combat this issue, I removed support for a lot of joy keywords. None of the keywords were vital for doing basic computation, they mostly revolved around looping and conditional statements. I would have liked to build a system that could handle parsing these errors, but I didn't have the time as my primary focus was getting a simulator that could start to solve some larger problems.

Resetting the Stack

When I started parsing output, I noticed something odd. The simulator would find programs that supposedly pass all of the test cases, but when I looked at the programs, it was clear that they shouldn't have passed any of them. After many hours spent debugging and trying to figure out why this was happening, I realized that since I was writing all the programs to a single file and running them at once, the joy stack wasn't being reset in between each program. This meant that any given program wasn't being run with just the input from the test cases, but also whatever data was left on the stack by the previous program, and the one before that, and so on. In order to fix this, I had to begin each program with "[] unstack". The unstack keyword in joy takes a list as an argument, and makes that argument the only thing on the stack. By passing it an empty list, the stack was reset to be empty again, and programs began to be scored correctly again.

Program Length

As I started attempting to tackle larger problems, I knew the simulator would need the freedom to create longer programs. I had a variable used to limit the maximum length a program could be (# of keywords), and I normally kept it around 25-30. For a more difficult problem, I would up this to 75-100, so the simulator would have more "space" to do various computations. However, when I did this, the simulator would occasionally crash, stating that it received no output from the joy interpreter. This led me to believe that something was happening with the joy interpreter itself when attempting to run these larger programs. Thanks to Dr. Wallingford, I found out that there was a hard limit set in the joy interpreter that would allocate memory for programs. Increasing this and recompiling my interpreter immediately fixed the issue.

Local Maximums

An unexpected challenge I came across was encountering local maximums. By chance, the simulator would often produce a program that scored well, but didn't have much room for improvement. It would be far from a perfect score, but the simulator lacked the variability to create programs diverse enough from the high scoring program to gain an improvement. I did two things to combat this, the first was to mess around with my simulator settings, particularly the mutation settings. I had initially arbitrarily chosen to have the top 10% of each population automatically move on to the next generation, and new programs would be created from that

10%. This lacked the variability needed to escape a local maximum. When I adjusted this to allow the top 50-60% of programs to move onto the next generation, I saw immediate improvement in the amount of generations that it took the simulator to reach any given score. The simulator now had the variability to escape the majority of local maximums eventually, but it still took quite a bit of time.

The second thing I did was add support to run multiple simulations at once, each on their own thread as to not impact performance. Because so much of evolution is pure luck in getting the right mutations, I would have wildly different results between each run of the simulation. Running multiple simulations at the same time helped fix this, so if one of the simulations got stuck at a local maximum 5 hours in, I wouldn't have just wasted those 5 hours, as I had several other simulations also running that whole time. Doing this was very demanding on system resources however, and I'm limited to running 10 simulations on my M1 Macbook Air (with 8GB RAM), and 15 on my desktop with a Ryzen 5 3600 (with 16GB RAM).

Results

Triangle Area

The first problem I attempted to solve with the simulator was finding the area of a triangle. This was the simplest problem that I could think of. The simulator was given six examples, it was given two input values, the base and height, and the expected output, the area. Starting with a population of 100, the simulator generally found a solution within 30-50 generations. I increased this gradually all the way up to 100,000, where a solution was usually found in the first or second generation.

Fibonacci Numbers

For a more difficult problem, and one I didn't expect the simulator to be able to solve, I gave it the goal of creating a program that can give me the nth fibonacci number. I gave it 30 test cases, where the inputs were the numbers 1-30, and the expected output was the corresponding fibonacci number. (1, 1, 2, 3, 5, 8, etc.) I used this problem when stress testing the system, as it would run for hours without reaching a solution, and would generate millions of different programs. This was useful when I was testing changes, as I could let it run for a while and see if any program it created managed to crash the system. I didn't expect it to be able to create a program that could generate fibonacci numbers, but it got extremely close. I let the simulator run for ~40 hours, and it was able to get relatively close, achieving a score of around 1 at the lowest. This was the program that achieved that: "dup dup dup dup dup dup dup 5 / sqrt - 7 / + 4 - dup 7 max 7 / + 4 - 7 / + 3 - 7 / + 4 - 3 max 8 - 7 / - 7 - 7 - 7 - 6 sqrt 4 rollup / - 7 / - 7 / + 4 - dup 6 sqrt - 7 / + 3 + 3 3 rollup / pow". I'm not going to pretend to understand this program, but when given each of the fibonacci numbers, its calculation is off by just a small fraction each time, so for the 30 test cases the difference totals up to be about 1.1680. This was significantly better than I expected, and I was thoroughly impressed. This specific program was generated

after only about 18-20 hours, and it didn't find any way to improve beyond it in the following 20 or so hours that the simulator ran.

Fuel Cost Problem

The fuel cost problem was used in another research paper on genetic programming, and was a part of their set of problems they used to analyze the performance of their system. The problem is as follows: Given n integers, divide each one by 3, round down, and subtract 2. Then sum the result of all the integers. This seemed to be a simple enough problem for my simulator to solve, although it would require the use of the "map" keyword to apply the operations to a variable number of integers. This was going to be the main challenge of this problem, and I wasn't sure if the simulator was going to be able to solve it or not. After letting the simulator run for 12 hours, I saw very little improvement in its score, and it failed to create any programs that effectively used the "map" keyword.

Conclusion

While the genetic simulator wasn't able to solve as many difficult problems as I hoped, I was impressed by how close it did manage to get, specifically on the fibonacci numbers. On top of that, I also gained lots of experience solving issues and debugging code, as a lot of my time this semester was spent doing exactly that. It seemed that each problem I solved introduced two new ones, but once I was able to get all the major problems solved and had a working simulator that I could mess around with, it was extremely rewarding and fun to tinker with.

If I had more time, I would have liked to add the ability to manipulate evolution settings while the simulator was running, to be able to see the immediate effect it would have on the system. From my experimentation, it appears that as the simulator gets a lower and lower score, it's harder for it to produce output that varies enough to improve the score. Being able to manipulate the mutation settings at this point to add more variability in programs generated would hopefully help the simulator escape any local maximums it encounters.

Overall, this was a very interesting and enjoyable project to work on. I was challenged regularly, and got to experience building a larger application than I had ever built in any other class during my time at UNI. Getting to work on a "large" (relative to other projects I've worked on) application I feel has prepared me a lot for the software engineering field, and entering into the workforce this summer where I'll likely be working on a project significantly larger than this one.