Genetic Programming

Creating Code Through Natural Selection

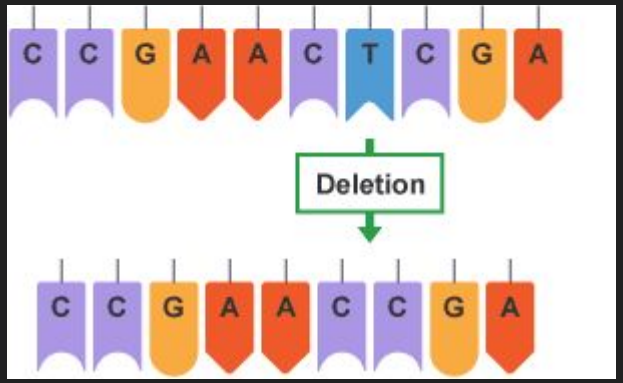
Project Overview

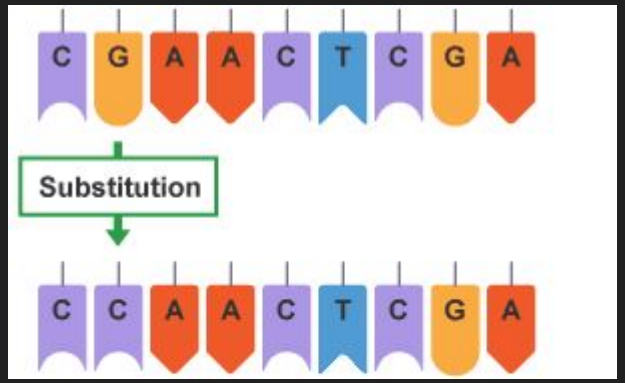
- Goal: Create a genetic programming simulator in C# for joy programs
- Why C#?
  - A language I'm already familiar with (Unity)
  - Faster than other languages I'm more comfortable with (Python)
    - Speed is important when running with large populations, more on that later…
- Why Joy?
  - Limited keywords
  - Stack based and concatenative, meaning any program can be written as a single string
  - Easy to mutate and crossover because it's concatenative
  - Interpreter is written in C - it's very fast
  - Ability to run multiple programs from a single file (Less file I/O = more speed)

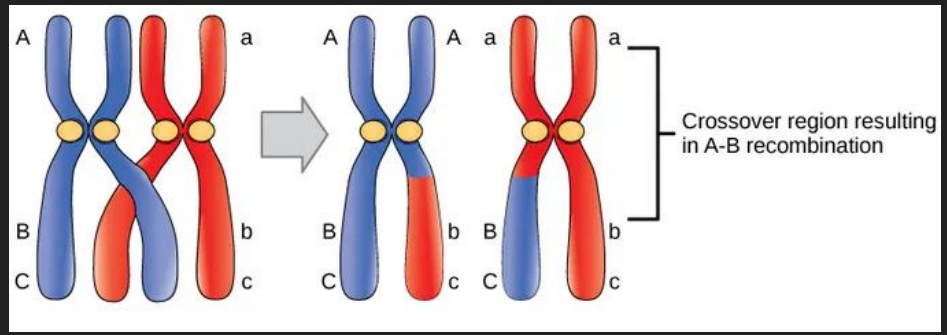
Genetic Programming Overview

- Creating programs through evolution (natural selection)
  - The user creates a set of test cases that consist of example inputs and their expected output
  - The simulation then creates a population of randomly generated programs that are run with the test case inputs (most of these won't run successfully)
  - A fitness function then assigns each program a score based on how close their output was to the expected output
  - A new population is then created by saving the best scoring programs, and creating new programs through crossover, mutation, and other evolutionary techniques
  - After many generations, a program should pass all the test cases, getting a perfect score

Biological Basis







Joy Programming Language

- A concatenative stack based postfix language
  - Concatenative:
    - Any set of joy programs can be concatenated to create one large program
  - Stack based:
    - The stack is the only way to store and manipulate data (No named variables)
    - All operators pull required number of argument(s) off the top of the stack
    - All operators push their result directly onto the top of the stack
  - Postfix:
    - Operators come last in programs
      - Ex: “A + B” is infix, “A B +” is postfix
      - Push A onto the stack, push B onto the stack, sum the first two elements on the top of the stack

Joy Programming Language Cont.

- Important operators (keywords)
  - All standard operators (+, -, *, /)
  - Most other basic operators (sqrt, pow, neg (negate), rem (remainder/modulo))
  - Special stack operators:
    - Dup - Makes a copy of the top element on the stack and pushes it to the stack
      - x -> x x
    - Swap - Swaps the top two elements of the stack
      - x y -> y x
    - Rollup - Moves items 2 and 3 to the top of the stack
      - x y z -> z x y
    - Rolldown - Moves top two items below the third item
      - x y z -> y z x

Simulator Overview/Settings

- Program flow:
  - Create population -> Run/score population -> Sort population -> Mutate population -> Run/score with new population (repeat)
- General Settings:
  - Population size
  - Max program length
  - Number of simulators
- Evolution settings:
  - Survival rate
  - Crossover rate
  - Mutation rate
  - Deletion rate

```
public static void Main(string[] args)
{
    int populationSize = 5000;
    int maxProgramLength = 75;
    int numSimulators = 10;

    List<Thread> threads = new List<Thread>();

    for (int i = 0; i < numSimulators; i++)
    {
        Simulator sim = new Simulator(
            populationSize,
            maxProgramLength,
            new JoyFitness(fibonacciNumbers),
            new JoyMutator(
                0.5f,  // Survival percentage
                0.7f,  // Crossover percentage
                0.25f, // Mutation percentage
                0.05f  // Deletion percentage
                ),     // The last three floats must add to 1
            i
            );
        Thread thread = new Thread(() => sim.Run(null));
        threads.Add(thread);
        thread.Start();
    }
```

```
public IGenome Run(int? stopGeneration){
    Logger.Log("Beginning simulation");
    Population = Mutator.CreateInitialPopulation(PopulationSize, MaxProgramLength);
    Logger.Log("Initial population created");
    while (true){
        RunGeneration();
private void RunGeneration(){
    ScorePopulation();
    SortPopulation();
    GenerationCount++;
    Population = Mutator.MutatePopulation(Population, PopulationSize, MaxProgramLength);
}
```

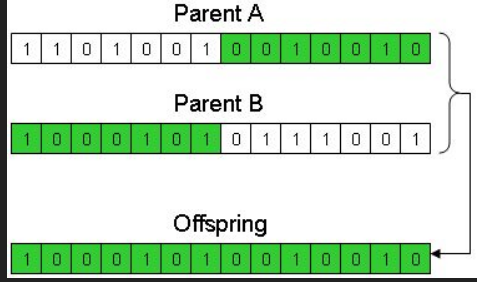
Scoring a Program

- For each test case, calculate the absolute value of the difference between the actual output and the expected output, and sum up all differences
- sum(abs(expected - actual) for each test case)
- When a program has a score of 0, it's passed all of the test cases

```
private static Dictionary<List<int>, float>
{
    { new List<int> { 2, 3 }, 3 },
    { new List<int> { 4, 5 }, 10 },
    { new List<int> { 6, 7 }, 21 },
    { new List<int> { 2, 5 }, 5 },
    { new List<int> { 1, 8 }, 4 },
    { new List<int> { 9, 4 }, 18 },
};
```

```
// Calculated as sum(abs(expected - actual) for each test case)
private float CalculateSingleScore(List<string> testCaseResults)
{
    float score = 0;
    try
    {
        List<float> resultList = testCaseResults.Where(s => !string.IsNullOrEmpty(s))
                                                .Select(float.Parse)
                                                .ToList();
        if (resultList.Any(float.IsNaN))
            return -1;
        for (int i = 0; i < resultList.Count; i++)
            score += Math.Abs(resultList[i] - TestCases.ElementAt(i).Value);
    }
    catch (FormatException)
    {
        return -1;
    }
    return score;
}
```

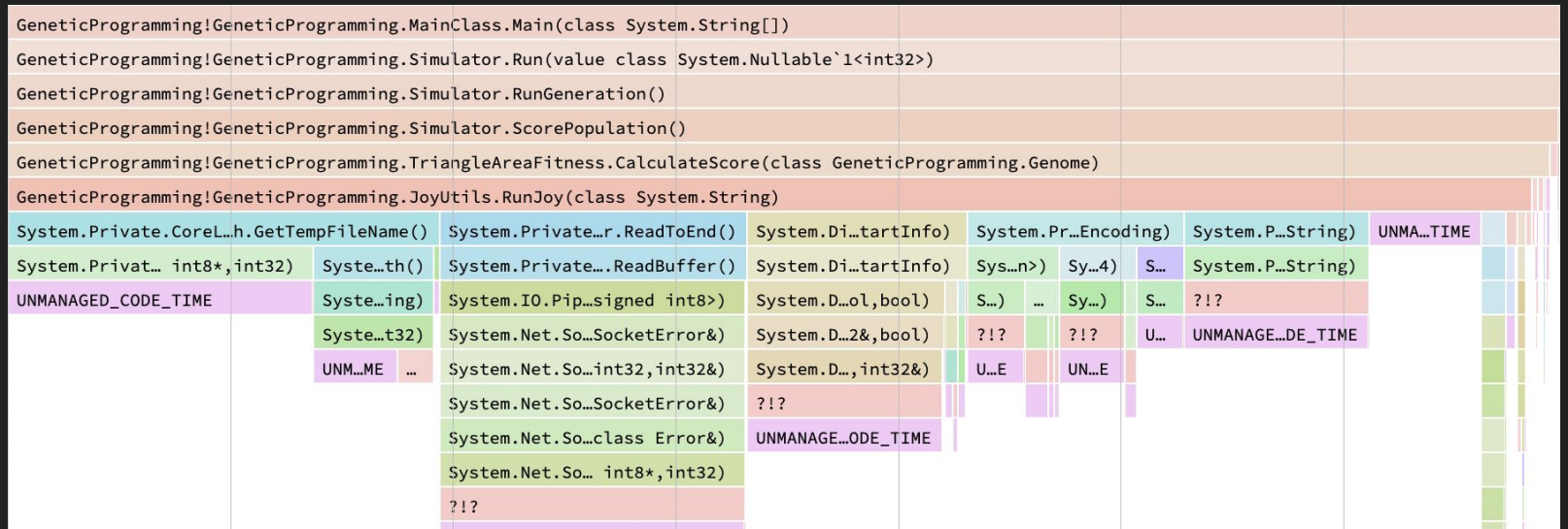
Evolving a Population



- Survival rate
  - Percentage of population that moves onto the next generation and is used for evolution
  - ~50-60% appears to work the best
    - Anything lower and there's not enough variation, anything higher and not enough new programs are created through evolution
- Evolution Settings
  - Crossover
    - Take two programs (the "parents"), and pick a random point on each one
    - Take everything after the point on the first program, and combine it with everything before the point on the second program, creating an entirely new program
  - Mutation
    - Pick a random token in a program, and change it to a different random token
  - Deletion
    - Delete a random token in a program
    - Useful for deleting portions of a program that don't contribute to the output

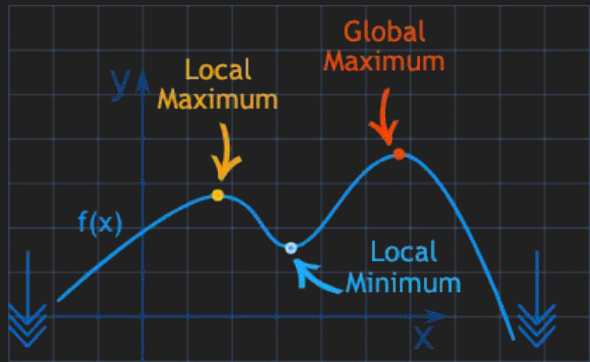
Challenges - Speed

- Initial simulator ran 30 generations with a population of 100 in ~2 minutes
  - Nowhere near fast enough for populations of 5,000+ that need to run for 10,000+ generations
- Rewrote fitness function to run a population of joy programs in bulk, saving significant amounts of time doing file I/O and creating subprocesses
  - Now runs the same benchmark in under 700ms



Challenges - Local Maximums

- Simulations would often get “stuck”, unable to improve their score after thousands of generations
- They lacked the variability to break out of a local maximum, and needed to make a large change in order to escape it
- Increasing the survival rate from 10% to 50-60% significantly reduced the chances that a simulation would get stuck at a local maximum
    - A larger variety of programs moved on to the next generation, allowing more variability in the mutations that took place.



Examples - Area of a Triangle

- Given the base and height of a triangle, calculate the area
  - (3 * 4) / 2 = 6
  - Joy: 3 4 * 2 /.
    - Push 3 then 4 onto the stack (Stack = [3, 4])
    - Multiply the top two numbers on the stack (Stack = [12])
    - Push the number 2 onto the stack (Stack = [12, 2])
    - Divide the second number on the stack by the first (Stack = [6])
    - Return the top of the stack

```
Beginning simulation
Initial population created
Thread: 0 Generation: 1 Best score: 19 Best program: 7 4 rotate - - *
Thread: 0 Generation: 2 Best score: 14.1847105 Best program: sqrt max 2 7 min *
Thread: 0 Generation: 3 Best score: 14.1847105 Best program: sqrt max 2 7 min *
Thread: 0 Generation: 4 Best score: 14.1847105 Best program: sqrt max 2 7 min *
Thread: 0 Generation: 5 Best score: 14.1847105 Best program: sqrt max 2 7 min *
Thread: 0 Generation: 6 Best score: 14.1847105 Best program: sqrt max 2 7 min *
Solution found in generation: 7
Solution: * 2 / ceil
```

Examples - Fibonacci Numbers

- Explicit formula for the nth fibonacci number (Binet's formula):

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

- Program generated after ~20,000 generations (~40 hours):
  - 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
  - Score: 1.168
  - First 10 fibonacci numbers generated by this program:
    - 0.7212, 1.167, 1.8905, 3.0607, 4.9953, 8.0225, 12.9845, 21.0094, 33.9941, 55.0036
  - Actual first 10 fibonacci numbers:
    - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

Examples - Fuel Cost

- Problem:
  - Given n integers, divide each by 3, round down, and subtract 2, then sum all the results together
  - Required the simulator to use the “stack” and “map” keywords to perform operations on a variable amount of input
- Results:
  - After ~12 hours, very little improvement was made, and no program made effective use of “map” or “stack”
  - Potentially too difficult for this system, as it needs to use “stack” to turn the input into a list, and “map” to apply an operation to each input. Incorporating both of these keywords would have no impact on the score until the correct function (divide by 3, round down, sub 2) is applied using map.
    - Adding some sort of incentive to use “map” and “stack” could produce better results

Questions?