Exploring how Genetic Algorithms perform on complex optimisation problems

Ryan Williams (17009972)

1 INTRODUCTION

The objective of this paper is to develop a Genetic Algorithm (GA) and benchmark it against difficult optimisation problems, notably the Rastrigin function and a modified Ackley function. This paper will also cover the development of a simple GA with selection, crossover and mutation. This GA will then be tested on well-known optimisation problems. The results will be averaged over as many runs as possible so that the results can confidently be used to draw conclusions about the algorithm as a whole.

Generally, the approach to this task was to develop the GA in such a way that allows us to programmatically change the selection type, fitness function, gene bounds etc. The GA was implemented so that with one function call a GA could be run hundreds or thousands of times, finally the mean and best fitness is average across every run for each generation and stored for analysis. This means that the results for each set of parameters is very consistent such that any randomness in any single run is minimised.

2 BACKGROUND RESEARCH

Ethical Issues

There are many ethical concerns surrounding the use of AI for problem solving, one of which is who is responsible for the AI when it fails. This is sometimes referred to as the "the problem of many hands" (Timmermans et al. 2010). Is the programmer accountable for the actions because it is their code? Or is the supplier of the data that the AI was trained from responsible?

To further this there has been many examples of human bias such as gender or race has been picked up by an AI. If at any point the data is manipulated by a human, the resulting AI will learn theses biases in training.

Evolutionary algorithms

Evolutionary algorithms are by no means a new concept, what was once pioneered by J. H. Holland in the 1960's has matured over the last 50 years. One reason for this is the computational power of hardware has increased exponentially and it is now much easier to develop and test such algorithms.

One Study (Man, Twang and Kwong 1996) states that this means that "hard" or even "impossible," in the past are no longer a problem as far as computation is concerned. As this study is over 20 years old I believe this is even more accurate since then as the development of faster and faster hardware has continued.

Optimisation is the task of finding solutions or values to a given problem that satisfy its boundaries or constraints. These constraints can be defined for any number of reasons such as a physical limitation in an engineering problem or in a business environment where cost, time and resources is a factor.

Optimisation problems are often very large in terms of search space, this means that to simply try every single possible value and find the best solution would take an unreasonable amount of time. This could be viewed as a brute force approach. However, this could be possible if the problem space is not large, with every additional parameter or variable that is defined the problem space becomes exponentially bigger. This is why optimisation algorithms are used, one problem with these algorithms is that they do not guarantee the best solution because they do not search every single solution. This can also happen if the algorithm gets stuck in what's known as a local optimum. This is when there is a good solution but not the best that the algorithm has to "move" away from to achieve a better solution.

As stated above one factor that makes these problems hard is that they contain many local optimums. That "trap" the EA meaning that it does not find the global optimum solution. This is reiterated in one study Bajpai and Kumar (2010) "classical optimization techniques have difficulties in dealing with global optimization problems" and that "One of the main reasons of their failure is that they can easily be entrapped in local minima"

Later in this study Bajpai and Kumar go on to suggest several different reasons why GAs as apposed to alternative EAs are good at global optimisation. In summary.

- GAs are intrinsically parallel, they explore many different solutions at a time (represented by an individual)
- Because of the previous point GAs are better suited to very large problem spaces.
- GAs perform well where there is many local optimum, it can escape local optimum.
- GAs can manipulate many parameters at a time (Mutation and Crossover)

The reasons listed above allow a simple GA to be able to achieve if not global optimum but a very good solution none the less to a modified version of the well know Rastrigin function that contains a large search space and has a lot of local minima.

3 EXPERIMENTATION

At a high level a GA is an algorithm that attempts to mimic evolution in the real world. This is done by having a set or "population" of "individuals" each of which has a set of values or "genes". These values are then used to decide the fitness of each individual. The gene values of an individual is used to calculate its fitness, the fitness of a individual is how good it is a solving a particular problem. Once the fitness of all the individuals is calculated, selection mutation and crossover occur. Selection happens first, it is where the offspring is selected as a sub set of high fitness individuals from the population. Mutation occurs after selection and is an, often small, random chance modifier used to mutate an individual's gene values. Finally, crossover is the process of selecting random "parent" genes that are used to create new individuals with some gene values from one parent and some from another. If there is no crossover or mutation then there is no way the best fitness can improve over every generation. The best solution that is randomly generated in the initial population is the best solution that the GA will be able to achieve.

One way to represent the genes of an individual is as binary. This means that to implement something like mutation it is very easy as you can simply flip the bit of a gene to mutate it. However representing the genes or solutions to a problem as a bit has its limitations. For example many real world problems require genes to be made up of real world values. This led me to modify my genes to be a float value in a set range. To do this first the individuals must be modified to contain a list of floating-point values. Then the selection process must be changed to be able to deal with negative values.

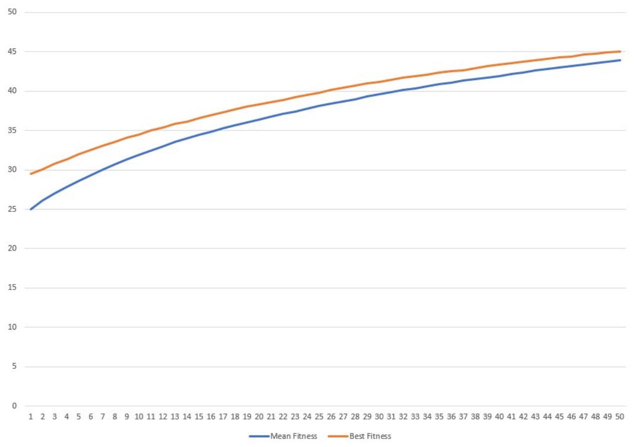

Figure01 – Adding fitness function (Population 50, Generations 50, Mutation rate 0.01 Mutation step 1)

After making these changes the above graph shows the average results of 100 runs of the GA with reseeded random. Note that the mean fitness starts at 25.03 as individuals with 50 genes initiated between 0.0 and 1.0 this makes sense. The best fitness achieved after 50 generations is 45.06. Increasing the number of generations that the GA runs to 250 allows the GA to reach the global maximum, 50 at generation 175. This fitness function is very simple and is more suited to testing the selection, mutation and crossover of a GA rather than benchmarking the performance.

Rastrigin function.

The Rastrigin function is a mathematical optimisation function that can be used to benchmark different algorithms. Finding the minimum of this function (0) is difficult as it has a large search space and it has a lot of local minima. To Benchmark the GA using the Rastrigin function some modifications must be made;

1. The gene values remain as float values but must remain in the range -5.12 to 5.12.
2. Both selection functions must be modified to work towards the minimum (for tournament selecting this is a simple as reversing the > operator.)
3. The best fitness must be modified to show the smallest fitness as this is a minimisation function.

As the gene values are already defined as floating point variables the only task was to modify the initial gene value generation to random select values between the defined bounds. The next task was to modify the selection process for both tournament and roulette. The tournament selection was modified so that after 2 individuals were selected the chosen individual was the individual with a fitness closest to 0. For roulette wheel selection it must be modified so that an individual with a fitness close to 0 has a bigger chance to be selected. This again is the exact opposite of before. Finally, when storing the results of the GA the best fitness is that which is closest to the known global optimum for this fitness function, 0.

After making these changes and running the GA 100 times with these values, population 50, number of genes 10 and mutation rate 0.01.

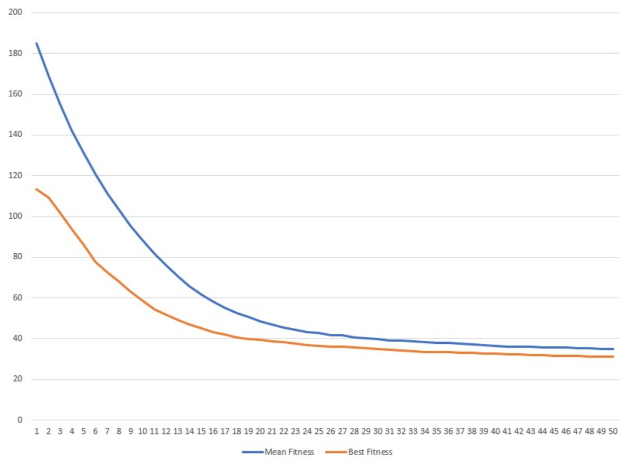

Figure 02 – Rastrigin function (Tournament Selection, Population 50, Generations 50, Mutation rate 0.01, Mutation step 1.0)

The Graph above shows that on average best fitness achieved by the GAs after 50 generations was ~31.1. After increasing the mutation rate to 0.03 the GAs achieved an average best fitness of 26.1, still very far from the global minimum 0. To give the GAs more time to achieve a better result I then increased the number of generations to 5000 whilst reverting the mutation rate to 0.01.

After 1000 generations the GAs only achieved a best fitness of 14.6 with only a slight improvement to 14.1 after another 4000 generations. This lead me to believe that they were getting "stuck" in a local minima, to combat this I then tried 2 different approcahes.

Firstly I tried several different combinations of mutations rate and mutation step (how much a gene is mutated by). I hoped that this would allow mutations that would change more of the genes in each individual and mutate them with a larger value meaning that solutions could mutate out of a local minima. However this was unsucessful and I did not mange to make any significant improvements.

Secondly I tried increasing the number of individuals in my population. This is so that instead of trying to mutate out of a local minima it would be less likely the GAs to fall into them in the first place. This is because I rely less on mutation and more on selection and crossover to select and maintain a good solutions and later mutation to make finer adjustments. In my mind this is because to have a larger population size the initial population covers more of the search space and therefore already contains better solutions that a lower population. This can be seen because the average best fitness of the first generation is 74.9 compared to 113.7 when the population is 5000 and 50 respectivly.

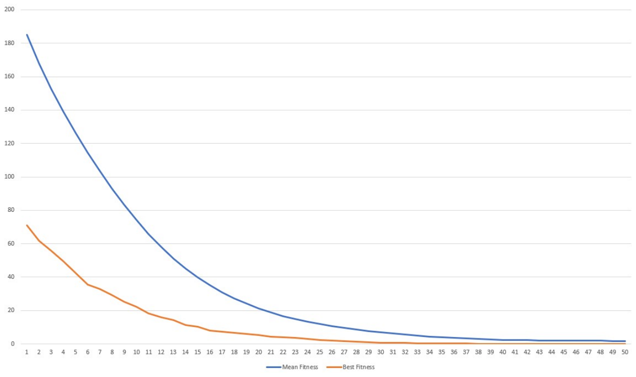

Figure 03 – Rastrigin function (Tournament Selection, Population 10000, Generations 50, Mutation rate 0.01, Mutation step 0.1)

After 50 Generations with a population size of 10000 and a low mutation rate and mutation step the averaged best fitness at generation 50 is 0.044, very close to global minimum.

To improve on this solution even more I increased number of generations to 10000 allowing the GAs longer to mutate once the solutions are approaching 0. I was hesitant to increase the Mutation rate however as I did not want to mutate my initial good solutions in the population. Therefore, the mutation rate was kept to 0.01 and the mutation step to 0.1. As before 10 separate runs with these values average a best fitness of 0.000363 after 10000 generations. This was replicated with roulette wheel selection however it provided worse results.

Modified Ackley function

Below is the same values for the GA but using a new fitness function, a modified version of the Ackley function. Similarly, to the previous fitness function I first started with a small population, a low mutation rate and 500 generations. This yielded the following results.

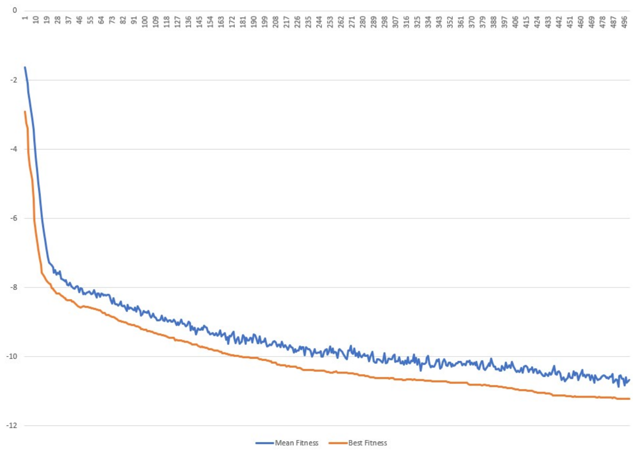

Figure 04 – Modified Ackley function (Tournament Selection, Population 50, Generations 500, Mutation rate 0.01, Mutation Step 1.0)

It should be noted that the above results are using a N (Number of genes) value of 10. This value can make the problem much easier or much more difficult as it increases the possible solution space. To improve on this solution, I applied the same steps as above on the Rastrigin function.

Firstly, I increased the number of generations, this did not work as the best fitness only improved by 0.6 after a further 4500 generations. This led me to believe yet again that the GAs were getting “stuck” in a local minimum. To combat this, I tried the same approaches as before, first increasing both the chance and size of the mutation before lowering it again and increasing the population a lot.

Neither increasing the Mutation rate and increasing the Mutations step whilst keeping the same number of generations and population made a significant improvement on the previous best fitness of -12.3. The next step was to try lowering the mutation rate and mutations step and allowing the GAs to run for as many generations as possible. However even after 100000 generations the best fitness achieved was only -10.01.

Some of the best results that I achieved was when I increased the population size to 10000 and lower the generations to 50 (with a high population the generations must be lowered to cater for run time) and the averaged best fitness at generation 50 was -22.52.

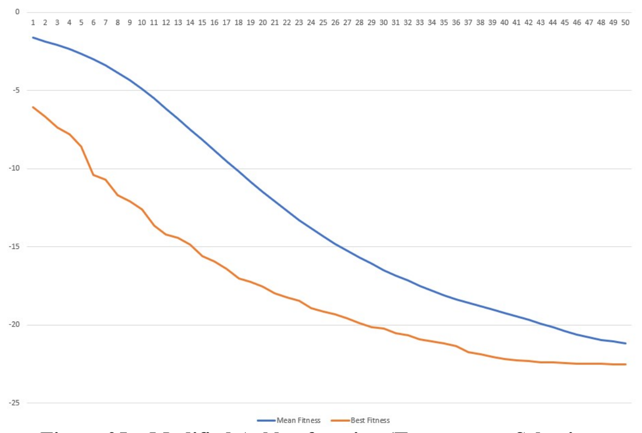

Figure 05 – Modified Ackley function (Tournament Selection, Population 10000, Generations 50, Mutation rate 0.01, Mutation Step 1.0)

Another option that I used to try and improve on this again is to use a differed selection method. Below are the same parameters except with the roulette wheel selection method, this again yielded slight worse results than tournament selection.

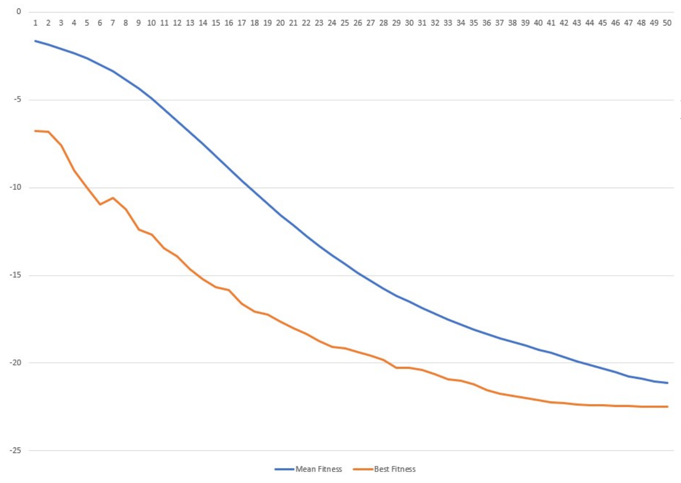

Figure 06 – Modified Ackley function (Roulette wheel Selection, Population 10000, Generations 50, Mutation rate 0.01 Mutation Step 1.0)

Finally, after increasing the number of generations to 500 and reverting to tournament selection the averaged best fitness reached was -22.6995.

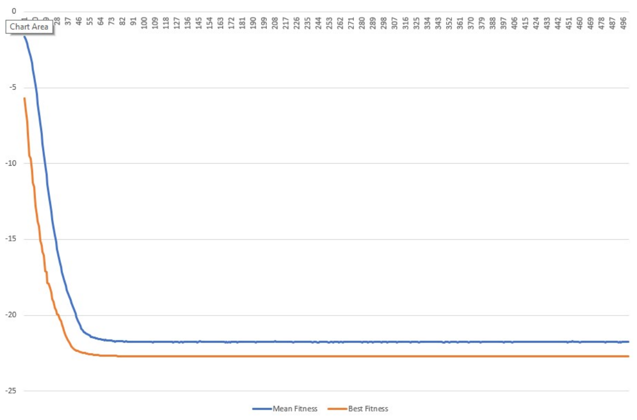

Figure 07 – Modified Ackley function (Tournament Selection, Population 10000, Generations 10000, Mutation rate 0.01 Mutation Step 0.1)

However, an optimisation problem with only 10 variables (10 genes) is not that difficult as the search space isn't that large. To see how the GA performed on a larger and more realistic problem I increased the N value to 100 and then 500. This meant that the GAs took significantly longer to run and reduced the speed at which I could make adjustments. The below graphs show just how much harder optimisation problems become when there are more variables to optimise.

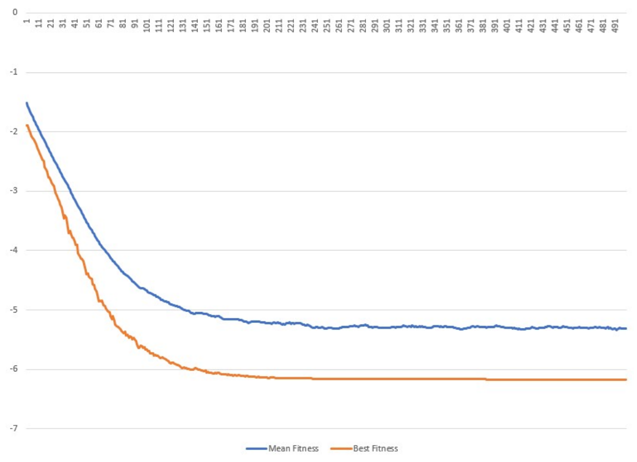

Figure 08– Modified Ackley function (Tournament Selection, Population 500, Generations 500, Mutation rate 0.01 Mutation Step 0.1) N = 100, Best fitness -6.1775

After this result I decided to use the same approach as before, firstly increasing the population size allowing more solutions to explore the search space simultaneously and secondly to experiment with the mutation rate and mutation step once a good result is achieved by just increasing the population. After increasing the population to 5000 the results are greatly improved, and the GAs averaged a best fitness of -14.51 at generation 500. As this made such a big improvement, I then used a population size of 25000. This again made a significant improvement and reached and average of -19.22 at generation 500, this is much closer to the best achieved when N =10 however it takes significantly longer to execute.

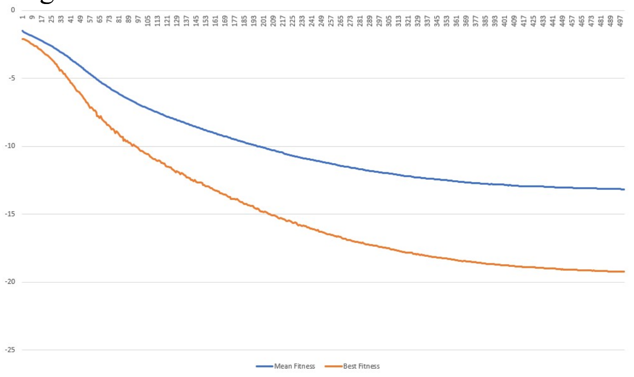

Figure 09– Modified Ackley function (Tournament Selection, Population 25000, Generations 500, Mutation rate 0.01 Mutation Step 0.1) N = 100, Best fitness -19.22.

Finally, I increased N to 500 to see how close I could get to the minimum achieved when N= 10.

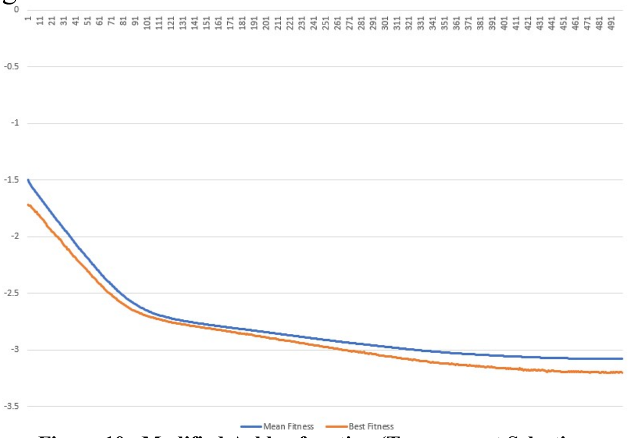

Figure 10– Modified Ackley function (Tournament Selection, Population 25000, Generations 500, Mutation rate 0.01 Mutation Step 0.1) N = 500, Best fitness -3.197

Even with a population of 25000 the best fitness is only -3.197. To achieve good results when the size of the problem space is so vast, I suspect I would have to have an even bigger population size and allow the GAs to run for many more generations. This would result in the GAs taking an unreasonable amount of time for benchmarking purposes. However, given the time and resources a GA such as this could solve even more complex problems with thousands of variables instead of just 500.

4 CONCLUSIONS

In conclusion much of the parameter tweaking that is required to obtain good results is repetitive and often difficult to predict. In future I would much prefer for this to be done programmatically. For example, if the mutation rate could be iterated through within set bounds or maybe the mutation values could go through their own selection, mutation and crossover process with their fitness being the best results of the GA. I think this could remove a lot of the tweaking that GAs need to be able to solve a specific problem.

REFERENCES

1. Bajpai, P. and Kumar, M., 2010. Genetic algorithm–an approach to solve global optimization problems. Indian Journal of computer science and engineering, 1(3), pp.199-206.

2. Eiben, A.E. and Smit, S.K., 2011. Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm and Evolutionary Computation, 1(1), pp.19-31.

3. Haupt, R.L., 2000, July. Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors. In IEEE Antennas and Propagation Society International Symposium. Transmitting Waves of Progress to the Next Millennium. 2000 Digest. Held in conjunction with: USNC/URSI National Radio Science Meeting (C (Vol. 2, pp. 1034-1037). IEEE.

4. K. A. De Jong, “Analysis of the Behavior of a Class of Genetic Adaptive Systems,” Ph.D. Dissertation, The University of Michigan, Ann Arbor, M1, 1975.

5. Man, K.F., Tang, K.S. and Kwong, S., 1996. Genetic algorithms: concepts and applications [in engineering design]. IEEE transactions on Industrial Electronics, 43(5), pp.519-534.

6. Pohlheim, H., 2007. Examples of objective functions. Retrieved, 4(10), p.2012.

7. Timmermans, J., Ikonen, V., Stahl, B.C. & Bozdag, E. 2010, "The Ethics of Cloud Computing: A Conceptual Review", IEEE, , pp. 614.

Source code as an appendix

Main.cpp
```cpp
#include <stdlib.h>
#include <iostream>
#include <time.h>
#include <random>
#include <iomanip>

#include "HelperDisplayGA.h"
#include "HelperBasicIO.h"
#include "GeneticAlgorithm.h"


int main()
{
	//BasicAdd Function
	TestGeneticAlgorithmLogResults(TOURNAMENT, BasicAdd, 100);
	TestGeneticAlgorithmLogResults(ROULETTE, BasicAdd, 100);

	//WS3 Fitness function
	TestGeneticAlgorithmLogResults(TOURNAMENT, WS3, 100);
	TestGeneticAlgorithmLogResults(ROULETTE, WS3, 100);

	//WOPT Function
	TestGeneticAlgorithmLogResults(TOURNAMENT, wOpt, 100);
	TestGeneticAlgorithmLogResults(ROULETTE, wOpt, 100);
}
```

GeneticAlgorithm.cpp
```cpp
#include <stdlib.h>
#include <iostream>
#include <time.h>
#include <random>
#include <iomanip>
#include <cmath>

#include "HelperDisplayGA.h"
#include "GeneticAlgorithm.h"
#include "HelperCSV.h"


float RandomFloat(float min, float max) {
	float random = ((float)rand()) / (float)RAND_MAX;
	float diff = max - min;
	float r = random * diff;
	return min + r;
}

float GenerateFitnessSimpleAdd(Individual ind)
{
	float fitness = 0;
	for (int i = 0; i < N; i++)
		fitness = fitness + ind.gene[i];
	return fitness;
}
```

```
float GenerateFitnessWS3(Individual ind)
{

	float fitness = 0;

	fitness += 10 * N;

	float x = 0.0f;

	float sum = 0.0f;

	float pie = atan(1) * 4;

	for (size_t i = 0; i < N; i++)
	{
		x = ind.gene[i];

		sum += (x * x) - (10 * cos(2 * pie * x));

	}

	fitness += sum;

	return fitness;

}

float GenerateFitnessWopt(Individual ind)
{

	float rv = 0.0f;
	constexpr double pi = 3.14159265358979323846;

	float firstSum = 0.0f;
	float secondSum = 0.0f;

	for (size_t i = 0; i < N; i++)
	{
		firstSum += ind.gene[i] * ind.gene[i];
		secondSum += cos(2 * pi * ind.gene[i]);
	}

	rv = -20 * exp(-0.2 * sqrt(float((1 / float(N)) * firstSum)));

	rv -= exp((1.0 / N) * secondSum);

	return rv;




	/*float rv = 0.0f;
	constexpr double pi = 3.14159265358979323846;
	float firstSum = 0.0f;
	double secondSum = 0.0f;
	double cosbit = 0.0f;

	std::setprecision(128);

	for (size_t i = 0; i < N; i++)
	{
		cosbit = 0.0f;
		firstSum += ind.gene[i] * ind.gene[i];
		cosbit = 2.0f * pi * ind.gene[i];
		secondSum += std::cos(cosbit * pi);
	}

	long double sqrt = (1.0 / N) * firstSum;

	long double exp1 = (-0.2f * std::sqrt(sqrt));
	rv = -20.0 * exp(exp1);

	double exp2 = (1.0 / N) * secondSum;
	rv -= exp(exp2);

	return rv;*/


}


float GetPopulationFitness(Individual population[])
{
	float t = 0;
	for (int i = 0; i < P; i++)
		t += population[i].fitness;
	return t;
}


float GetBestFitnessInPopulation(Individual pop[])
{

	float returnValue = FLT_MAX;
	//float wv;
	for (size_t i = 0; i < P; i++)
	{
		////convert
		//wv = 0;
		//wv = pop[i].fitness;
		//if (wv < 0.0f)
		//{
		//	wv = wv * -1;
		//}


		if (pop[i].fitness < returnValue)
		{
			returnValue = pop[i].fitness;
		}
	}
	return returnValue;
}

float GetMeanFitnessInPopulation(Individual pop[])
{
	float returnValue = 0.0f;
	for (size_t i = 0; i < P; i++)
	{
		returnValue += pop[i].fitness;
	}
	return returnValue / P;
}


/// <summary>
/// Runs a GA
/// </summary>
/// <param name="selectionType">
/// enum defined in GeneticAlgorithm.h
/// </param>
/// <returns>
/// Returns a list of GenerationResult objects, each one contains the mean and best fitness 
values
/// </returns>
GeneticAlgortihmResult RunGeneticAlgorithm(SelectionType selectionType, FitnessFunction 
fitnessFunction, float migv, float magv)
{
	GeneticAlgortihmResult returnValue;
	std::vector<GenerationResult> generationResults;

	//GA Result contains every generation and its mean fitness

	float mingv = migv;
	float maxgv = magv;



	switch (selectionType)
	{
	case ROULETTE:
		switch (fitnessFunction)
		{
		case WS3:
			printf("Benchmarking GA with RW selection and ws3 (Pop %d, Gene size %d, 
Generations %d, Mutation rate %f)...\n", P, N, GENERATIONS, MUTRATE);
			mingv = -5.12f;
			maxgv = 5.12f;

			break;
		case wOpt:
			printf("Benchmarking GA with RW selection and wopt (Pop %d, Gene size 
%d, Generations %d, Mutation rate %f)...\n", P, N, GENERATIONS, MUTRATE);
			mingv = -32.0f;
			maxgv = 32.0f;
			break;
			case BasicAdd:
				printf("Benchmarking GA with RW selection and Basic Add (Pop %d, Gene
size %d, Generations %d, Mutation rate %f)...\n", P, N, GENERATIONS, MUTRATE);
				break;
			default:
				break;
			}
			break;
		case TOURNAMENT:
			switch (fitnessFunction)
			{
			case WS3:
				printf("Benchmarking GA with Tournament selection and ws3 (Pop %d, Gene
size %d, Generations %d, Mutation rate %f)...\n", P, N, GENERATIONS, MUTRATE);
				mingv = -5.12;
				maxgv = 5.12;
				break;
			case wOpt:
				printf("Benchmarking GA with Tournament selection and wopt (Pop %d, Gene
size %d, Generations %d, Mutation rate %f)...\n", P, N, GENERATIONS, MUTRATE);
				mingv = -32.0f;
				maxgv = 32.0f;
				break;
			case BasicAdd:
				printf("Benchmarking GA with Tournament selection and Basic Add (Pop %d,
Gene size %d, Generations %d, Mutation rate %f)...\n", P, N, GENERATIONS, MUTRATE);
				break;
			default:
				break;
			}

			break;
		default:
			break;
		}

		Individual population[P];
		Individual offspring[P];

		/////////////////////////////
		//Create initial population//
		/////////////////////////////
		for (int i = 0; i < P; i++)
		{
			for (int j = 0; j < N; j++)
			{
				population[i].gene[j] = RandomFloat(mingv, maxgv);
			}
			population[i].fitness = 0;
		}

		/////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
		///////////////////////////////////////////////////////////////////////////////main
loop//////////////////////////////////////////////////////////////////////////////
		/////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

```
	for (int i = 0; i < GENERATIONS; ++i) {

		GenerationResult currrentGeneration;

		currrentGeneration.generation = i + 1;


		//Get fitness values for each individual
		for (int j = 0; j < P; j++)
		{
			switch (fitnessFunction)
			{
			case BasicAdd:
				population[j].fitness = GenerateFitnessSimpleAdd(population[j]);
				break;
			case WS3:
				population[j].fitness = GenerateFitnessWS3(population[j]);
				break;
			case wOpt:
				population[j].fitness = GenerateFitnessWopt(population[j]);
				break;
			default:
				break;
			}

		}

		/////////////
		//Selection//
		/////////////
		switch (selectionType)
		{
		case ROULETTE:
		{

			for (int individual = 0; individual < P; individual++)
			{
				float selection_point = RandomFloat(0, 
GetPopulationFitness(population));
				float running_total = 0;
				int r = 0;
				while (running_total <= selection_point) {
					running_total += population[r].fitness;
					r++;
				}
				offspring[individual] = population[r - 1];
			}

		}
		case TOURNAMENT:
		{
			for (int i = 0; i < P; i++) {

				int parent1 = rand() % P;
				int parent2 = rand() % P;

				if (population[parent1].fitness < population[parent2].fitness)
```

```
                        offspring[i] = population[parent1];
                    else
                        offspring[i] = population[parent2];
                }
                break;
            }
            default:

                break;
            }

            /////////////
            //Crossover//
            /////////////
            Individual temp;
            for (int i = 0; i < P; i += 2)
            {
                temp = offspring[i];
                int crosspoint = rand() % N;
                for (int j = crosspoint; j < N; j++)
                {
                    offspring[i].gene[j] = offspring[i + 1].gene[j];
                    offspring[i + 1].gene[j] = temp.gene[j];
                }
            }

            ////////////
            //Mutation//
            ////////////
            for (int i = 0; i < P; i++)
            {
                for (int j = 0; j < N; j++)
                {
                    float chance = RandomFloat(0.0f, 1.0f); //MAX MUTRATE IS 100
                    if (chance <= MUTRATE)
                    {
                        //Should i be checking bounds
                        float alter = RandomFloat(0.0, MUTSTEP);
                        if (rand() % 2)
                        {
                            if (offspring[i].gene[j] + alter <= magv) //not
going to go out of bounds
                                offspring[i].gene[j] = offspring[i].gene[j] +
alter;
                            else
                                offspring[i].gene[j] = maxgv; //Clip to max
                        }
                        else
                        {
                            if (offspring[i].gene[j] - alter <= mingv) //Dont
let it outside bounds
                                offspring[i].gene[j] = offspring[i].gene[j] -
alter;
                            else
                                offspring[i].gene[j] = mingv; //Clip to min
                        }
                    }
```

```
				}
			}

			///////////////////////////////////////
			//Calculate and store fitness values//
			///////////////////////////////////////
			currrentGeneration.bestFitness = GetBestFitnessInPopulation(population);
			currrentGeneration.meanFitness = GetMeanFitnessInPopulation(population);

			//returnValue.push_back


			generationResults.push_back(currrentGeneration); //Push this generation onto
the return value

			/////////////////////
			//Update Population//
			/////////////////////
			for (int i = 0; i < P; i++)
			{
				population[i] = offspring[i];
			}

		}

		returnValue.GenerationResults.assign(generationResults.begin(),
generationResults.end());

		return returnValue;
}


void TestGeneticAlgorithmLogResults(SelectionType selectionType, FitnessFunction
fitnessFunction, int numberOfRuns)
{

		srand(time(NULL));

		std::vector<GeneticAlgortihmResult> result; //Store result of the GAs
		GeneticAlgortihmResult allRunsAveraged; //Final result for each GA averaged

		float meanFit = 0.0f, bestFit = 0.0f; //used for averaging

		//Run GA numberOfRuns times, store result each time
		for (size_t i = 0; i < numberOfRuns; i++)
			result.push_back(RunGeneticAlgorithm(selectionType, fitnessFunction, 0.0f,
1.0f));


		printf("Averaging results for each generation...\n");
		//for every generation...
		for (size_t j = 0; j < GENERATIONS; j++)
		{
			GenerationResult allGensAveraged;  //Final result for each generation averaged
			meanFit = 0.0f; //reset each time
			bestFit = 0.0f; //reset each time
			//for every time we ran the GA...
			for (size_t i = 0; i < numberOfRuns; i++)
			{
				meanFit += result[i].GenerationResults[j].meanFitness;
				bestFit += result[i].GenerationResults[j].bestFitness;
			}

			//average
			meanFit = meanFit / numberOfRuns;
			bestFit = bestFit / numberOfRuns;

			//store vals
			allGensAveraged.generation = j + 1;
			allGensAveraged.meanFitness = meanFit;
			allGensAveraged.bestFitness = bestFit;

			allRunsAveraged.GenerationResults.push_back(allGensAveraged);
		}

		//write to console then csv
		PrintGAResultToConsole(selectionType, allRunsAveraged);
		WriteGAResultToCSV(selectionType, fitnessFunction, allRunsAveraged);

}
```

GeneticAlgorthm.h

```
#pragma once

#include <vector>


const int N = 500; //Number of genes
const int P = 25000; //Population
const int GENERATIONS = 500; //Generations

const float MUTRATE = 0.01;
const float MUTSTEP = 0.1;

//Each individual
typedef struct {
	float gene[N];
	float fitness;
} Individual;

//Each generation
typedef struct {
	int generation;
	float bestFitness;
	float meanFitness;
} GenerationResult;

//Each time we run
typedef struct {
	std::vector<GenerationResult> GenerationResults;
} GeneticAlgortihmResult;
```

```
enum SelectionType { ROULETTE, TOURNAMENT };

enum FitnessFunction { WS3, wOpt, BasicAdd};

void TestGeneticAlgorithmLogResults(SelectionType selectionType, FitnessFunction
fitnessFunction, int numberOfRuns);
```