**The Effects of Parameter Changes on Comparable Evolutionary Algorithms**

22019799

# Introduction

Set out to find the optimal configuration for an Evolutionary Algorithm (EA), the experiment attempts to understand each argument and its resultant effect on the solution of the fitness. A well-reasoned thought might suggest trying random numbers, and extrapolating tests from initial findings. That is not an efficient solution, nor is it effective. Instead, this experiment focuses on an underlying Meta program which can incrementally test every value in a range, presenting the best results.

The same form of experimentation will be carried out on two different minimisation equations. With the project’s backbone consisting entirely of C++, it would be able to store all the necessary data in relevant object types. This ensures that all tests can be independently explored and processed accordingly.

These results are then plotted graphs using an open-source GUI library, allowing them to be further analysed by an operator. This process helps to develop a deeper understanding on the effects of the parameter changes.

A small portion of the pattern can be observed by first recognising that the solution’s fitness increases with the tournament size. The opposite could be said for both Mutation Rate and the Mutation Height: as these numbers grow, the solution’s fitness shrinks dramatically.

# Experimentation

Evolutionary Algorithms typically begin with a pre-defined set of data to manipulate. Given that the experiment is entirely focused on the effects of the parameters belonging to the EA, using pseudo random genes would be equally acceptable. To determine if an individual’s genes hold any merit: a fitness scoring function must be used.

After scoring the whole population, next step is to feed these individuals into a Tournament Selection process. The basis for this comes from the most known examples of the Roman Empire. Gladiator fights were common practice and served a way for spectators to gather and watch as two warriors fought to the best of their abilities. This method of fighting gave way to a form of forced natural selection, exemplified by the first stage of the EA. Initially, the algorithm should loop through the population, choosing a champion each time and selecting the other challengers from the population at random. In each gladiator battle, only one can be the victor. This is determined by checking who, of the current gladiators, has the highest fitness score. The victor is then passed onto a new list of individuals, moving the cycle onto the next champion in the population.

Once the list of champions is complete, the Crossover function is initiated. As the name may suggest, it draws inspiration from the molecular biology method of DNA modification known as Gene Splicing. In a rough outline, this technique involves cutting a piece of the subject’s DNA from one enzyme and replacing it with an identical cutout from another, more admirable enzyme. A similar process is attached to this function. Unlike the Gladiator battle, this function depends on visiting each individual exactly once. To accomplish this, the population loops through every other member and selects the next member by simply indexing the next individual in the population. Here, a random number between zero and the number of defined genes is generated. This number is then treated as a point in the gene list, after which the remaining genes are transferred to the other individual. Both individuals have their fitness recalculated, and then are transferred to a new population list.

Finalising the Crossover function is imperative for the Mutation Sequence to begin. Once again using biology as a backbone, more specifically the fields of virology and evolutionary biology, mutation serves to introduce chaos, or chance, into an algorithm. Inside the function, each individual is once again looped, and so are all of their genes. For each gene, a random first number is generated. If this number is lower than a predefined Mutation Rate, a second random number is generated. The second number will be between the positive and negative values of the Mutation Height, another predefined entity. This signed variable is then added to the gene. However, if this addition moves the gene beyond the accepted upper, or lower, thresholds: the gene is instead replaced with the accepted threshold. After the genes have all had the chance to mutate, the fitness is recalculated for a final time, and the population is copied to a new list.

On completion of the population alterations, only one step remains. Evolution is the driving field behind this type of search algorithm, and focuses on the contributions made by Darwinism. In natural selection, the individuals with the best features are more likely to survive. Over generations, this repeated form of natural selection adapts the individual to fit their environment. The exact same process can be forced onto an EA, this is the method called Elitism. In the realm of Artificial Intelligence, this selection process involves surveying each population’s fitness mean average. The most favourable individuals are passed through to the next generation: restarting the cycle.

Different strengths of elitism could be applied, a simple pass-down would ensure that all alterations are passed on to the next generation, disregarding the need for multiple populations. However, this holds potential to result in backsliding, decreasing the population’s average fitness over time. More sophisticated options would see each population tested. This, too, would be a mistake. Instead of incorporating mutations every time the population advances, they would only be included if the mutated population scores higher than the others.

Since Mutations have a 50% chance of being a negation from the altered gene, this also means that: because the fitness could remain the same, the chance of the mutations being carried over is even less. That leaves one last option: only the final population is weighted against the initial population. The victor proceeds. This ensures that if any progress has been made at all, the mutations created will also be passed down, ensuring the algorithm never hits a brick wall.

As previously stated, plugging random numbers into the positions of mutation height, mutation rate, and tournament size is not optimal. It would take too long, and automating the process is far more efficient. Three nested loops run, incorporating the described Evolutionary Algorithm. At the end of each test, a data structure is created which includes: the current test’s parameters, the ending fitness average, and the solution’s fitness rating. The solution’s fitness rating is calculated by multiplying the final ending fitness by the number of loops performed.

The final piece of data stored in the new structure is the generation at which the increase in fitness stops changing or plateaus. This is crucial for calculating the overall fitness of the solution, because a lower generational plateau would indicate a solution is found quicker. Ensuring the parameter search finds the most optimal solution is vital; without factoring in the plateau, this is not possible.

Once the meta-data list of tests has concluded, graphs are available for viewing. The combination of these different graphs allow for an easy understanding of the cause-effect relationship existing between the arguments inserted to the algorithm, and the efficiency of the results. A key detail in the determination of an argument’s outcome, however, is also the fitness calculation used. Two minimisation equations were used to compare results which are applicable for general, and specific models.

Knowing how these nested loops are structured is one of the most important things in understanding the graphs’ presentation. On the outside, there is a loop testing the Tournament Size (Represented by %S) by checking every value between the current systematic search bounds. The second loop is structured to test the bounds of the Mutation Rate (Represented by %MR), followed by a loop exploring the Mutation Height (Represented by %MH) bounds.

The loops are simple. Each time, they run the Evolutionary Algorithm; arguments increase incrementally. In each test, the Tournament Size is increased by one with each iteration. Both the Mutation Rate and Mutation Height are consistently increased by 0.01.

|  |  |  |
| --- | --- | --- |
| Operator Variable | Large Test | Medium Test |
| Tournament Size (%S) | 2 -> 10 | 2 -> 10 |
| Mutation Rate (%MR) | 0.01 -> 0.5 | 0.01 -> 0.3 |
| Mutation Height (%MH) | 0.01 -> 0.5 | 0.01 -> 0.3 |

(Article 0 – Meta argument search list)

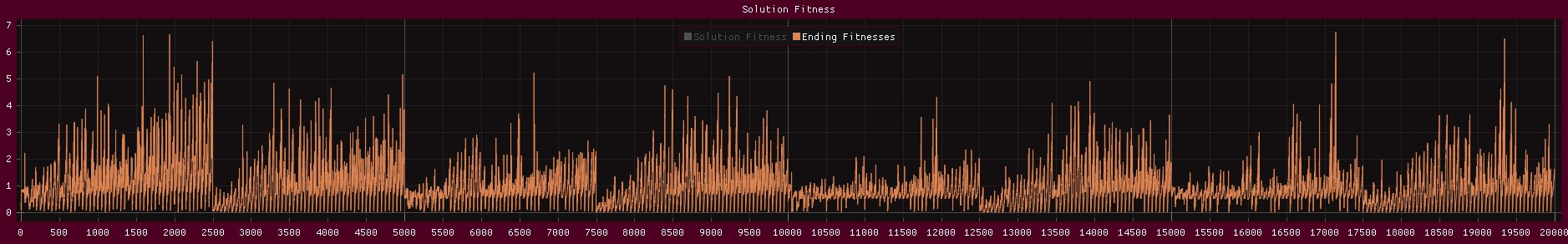
Before the graphs can be explained in their entirety, the gene fitness calculations must be defined. A couple of math equations

Description automatically generated with medium confidence

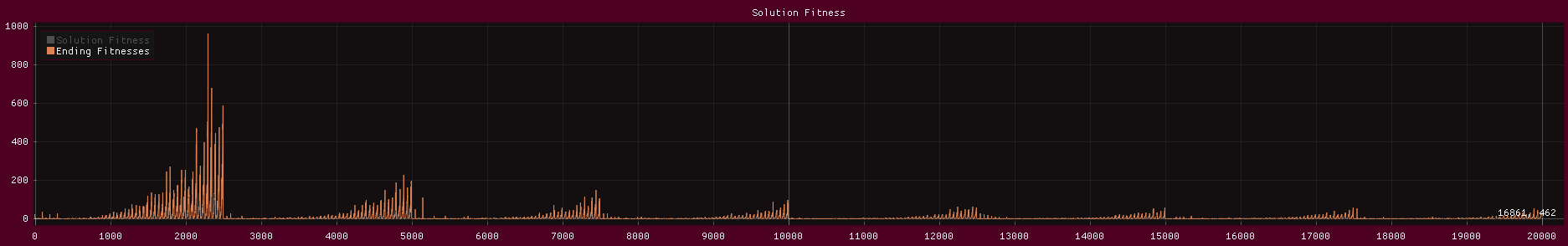
(Figure 0 – Minimisation functions one and two)

These two functions are vastly different and have different advantages to both. The real power, however, is combing the output of them both and cross referencing the best results. While minimisation function one appears to be the most simplistic, it is the most inefficient for computing power. This must be factored in when trying to select a model to use, an inefficient model would cause the algorithm to draw in much more computing power than necessary. Over 100 tests, the first equation completes 100 generations and averaged 0.0186s. The second equation averages 0.0135s.

Though this difference may seem inconsequential, its important to cascade the buffer’s impact when servicing potentially thousands of requests. This buffer’s primary cause is the difference in the position of the squared number in the equation. Instead of only having one inside a loop, there are two. Simply speaking, this near doubles the number of calculations needed because of how computers handle mathematics.



(Figure 1- Graph of Ending Fitness of Minimisation One – Large Study)

(Figure 2 - Graph of Ending Fitness of Minimisation Two – Large Study)

As previously discussed, there are some things which are argument dependant. Others are minimisation dependant. A resultant can easily be determined as being controlled by the minimisation function by cross referencing the function’s graphs. If similarities are established, they may be argument dependant. Otherwise, they are most likely to be controlled by the fitness function. Clear evidence of a case, such as this, is that: whilst in Figure 1 there is an obvious relationship between the tournament size being even, and the consistency of the ending fitness. This relationship is not apparent with the second function.

On the opposite side of the spectrum, the first thing that should be noticed as a simple cause-effect relationship: whilst the tournament size increases, the ending fitness average decreases. Although this is true, it is important to note the drawback. Increasing the tournament size exponentially increases the number of loops needed to complete the tournament. If this subsequently passes a threshold; the solution may reach a better ending fitness, but it would demand too many computing resources. As the tournament size increases, obviously so do the number of randomly selected Challengers in the tournament. This does also mean that the chance for an individual to be duplicated by winning more than one tournament increases, allowing for a quicker repopulation of higher value individuals.

Initial findings would suggest that striking the lowest tournament selection options could be a good idea. This would, in turn, remove the higher spikes from the parameter search. It is important to first note the reason for the spikes in the ending fitness average.

A screen shot of a computer

Description automatically generated

(Figure 3 – Zoomed in Figure 1, **Generations** 0 -> 2500, **%S** = *2*)

A screen shot of a game

Description automatically generated

(Figure 4 - Zoomed in Figure 2, **Generations** 0 -> 2500, **%S** = *2*)

A graph on a black background

Description automatically generated

(Figure 5 – Zoomed in Figure 2, **Generations** 0 -> 600, **%MR** *0.01 -> 0.12*)

Leading onto the second relationship established: when the Mutation Rate is increased, the ending fitness height becomes less desirable. The rise of the Mutation Rate means that the gene pool holds less integrity, and eventually becomes unstable. Eventually this results in the complete loss of admirable genes, as represented by Figure 3 and 4. While this does consistently reduce the time taken for the algorithm to plateau, the ending result is significantly worse.

A graph on a screen

Description automatically generated

(Figure 6 – Zoomed in Figure 1, **Generations** *300 -> 350*, **%MR** = *0.07*, **%MH** *0.01 -> 0.5*)A screen shot of a graph

Description automatically generated

(Figure 7 – Zoomed in Figure 2, **Generations** *300 -> 350*, **%MR** = *0.07*, **%MH** *0.01 -> 0.5*)

Like Mutation Rate, Mutation Height has a simple cause-effect relationship. When the height rises, the fitness once again becomes less desirable. The same precedent holds true; although in this case, the loss of integrity is much steeper. Ultimately, this results in the ending fitness being exponentially high, especially when combined with a higher Mutation Rate. Using both, a high Mutation Hight and Mutation Rate, would cause a phenomenal reduction in how good the output is. For example, when **%MR** and **%MH** both equal 0.5: roughly every other gene is being altered. This could be altered by anywhere between -0.5 and +0.5.

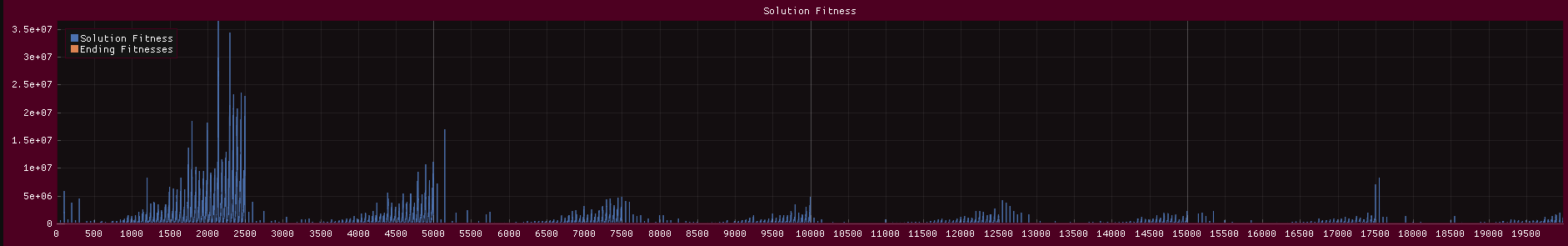
The synergy of a large Mutation Height and Mutation Rate have already been exposed. Similar reactions occur when faced with the combination of a low value for each. Opposite to the complete loos of integrity, the gene pool now holds too many original values. Either taking too many generations to reach a plateau, or not finding one at all; this can result in not finding a desirable outcome, causing the algorithm to plateau at the original lowest value.

Discussing the difference between a good solution and an optimal solution is only possible once the efficiency is first calculated. There have been reviews of the negative effects of incorporating specific parameters. Using this knowledge, it is possible to calculate a formula which totals the number of loops preformed by the Evolutionary Algorithm, and multiply it by the ending fitness. This would cause any fitness below 1 to act as a divisor, reducing the total solution fitness rating. The opposite would happen when the ending fitness rests above 1: pushing a bigger gap between the optimal and poor solutions. That would, in turn, force the most optimal solutions to present themselves by factoring in computing resources needed to find an equal ending fitness.

A screen shot of a graph

Description automatically generated

(Figure 8 – Graph of Solution Fitness Rating for Minimisation One – Large Study)



(Figure 9 - Graph of Solution Fitness Rating for Minimisation Two – Large Study)

As discussed, the spikes apparent on Figures 8 and 9 are a clear indication of times where the solution either didn’t plateau or reach a desirable result. Using the generational inspection tool, it is easy to see which parameters should be clamped in the meta search. The biggest cause for the higher spikes: the Mutation Rate and Height are over 0.3.

A screen shot of a graph

Description automatically generated

(Figure 10 – Graph of Solution Fitness Rating for Minimisation One – Medium Study)

A screen shot of a computer

Description automatically generated

(Figure 11 – Graph of Solution Fitness Rating for Minimisation Two - Medium Study)

As evidenced by the stark contrast between the two studies, removing the higher spectrum arguments quickly gives a multitude of more optimal solutions and keeps the results within much closer boundaries. Shown by the remaining spikes and previous statements, something else that will regularly decrease the optimisation of a solution: the lower Mutation Rate and Height combination.

Small alternations to the next round of testing, include altering the minimum limit for the mutation height. This takes advantage by retaining optimal solutions, while discarding the insignificant.

|  |  |
| --- | --- |
| Operator Variable | Inspection |
| Tournament Size (%S) | 3 -> 10 |
| Mutation Rate (%MR) | 0.04 -> 0.25 |
| Mutation Height (%MH) | 0.04 -> 0.2 |

(Article 0 – Meta argument search list)

Slowly reducing the number of insignificant test parameters is one of the best ways to understand the cause effect relations which exist between them, and the optimality of the final solution. After whittling down the systematic search, there comes a final graphing stage. Also produced is a table including the ten best solutions, and the ten most optimal solutions. These can be used to directly view the results of the search.

A screenshot of a graph

Description automatically generated

(Figure 12 – Graph of Ending Fitness for Minimisation One, Inspection Study)

A screenshot of a computer

Description automatically generated

(Figure 13 – Graph of Solution Fitness for Minimisation One, Inspection Study)

A screen shot of a graph

Description automatically generated

(Figure 14 – Graph of Ending Fitness for Minimisation Two, Inspection study)

A screen shot of a computer

Description automatically generated

(Figure 15 – Graph of Solution Fitness for Minimisation Two, Inspection study)

As proven by the upper and lower bounds of the graphs, presented in figures 12-15: the predefined relationships between the Tournament Size, Mutation Height and Mutation Rate hold true. They can be used to help predict better results by knowing which specific are used. With this, the ending result caps out at 1.1 for Minimisation One, and 0.4 for Minimisation Two. In the original round of testing, the boundaries max out at 6.9 for Minimisation One and 100 for Minimisation Two; representing an 84% and a 996% increase in the lowest outputs for Minimisation one, and Minimisation Two.

Having removed any undesirable solutions, there are now ten clear solutions which will consistently reach higher results. From the solution fitness rating, it is also possible to easily select the ten most optimal solutions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| %S | %MR | %MH | Ending Fitness | Plateau | Solution Fitness Rating |
| 7 | 0.05 | 0.04 | 0.0049923961 | 41 | 317.2667846680 |
| 7 | 0.06 | 0.05 | 0.0097258165 | 41 | 318.0756225586 |
| 7 | 0.05 | 0.05 | 0.0074842549 | 39 | 452.4232177734 |
| 9 | 0.07 | 0.05 | 0.0083729550 | 41 | 532.1013183594 |
| 7 | 0.07 | 0.04 | 0.0080959741 | 52 | 652.5355224609 |
| 9 | 0.05 | 0.05 | 0.0100589141 | 47 | 732.7773437500 |
| 9 | 0.07 | 0.04 | 0.0096950391 | 58 | 871.5840454102 |
| 9 | 0.07 | 0.07 | 0.0245385002 | 25 | 950.8668823242 |
| 7 | 0.11 | 0.06 | 0.0240848400 | 30 | 1119.9450683594 |
| 9 | 0.04 | 0.04 | 0.0106604937 | 68 | 1123.61600888672 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| %S | %MR | %MH | Ending Fitness | Plateau | Solution Fitness Rating |
| 7 | 0.05 | 0.04 | 0.0049923961 | 41 | 317.2667846680 |
| 7 | 0.05 | 0.05 | 0.0074842549 | 39 | 452.4232177734 |
| 7 | 0.07 | 0.05 | 0.0080959741 | 52 | 652.5355224689 |
| 9 | 0.07 | 0.04 | 0.0083759550 | 41 | 532.1013183594 |
| 9 | 0.07 | 0.05 | 0.0100587141 | 47 | 732.7773437500 |
| 9 | 0.04 | 0.05 | 0.0109889492 | 62 | 1048.3499755859 |
| 9 | 0.05 | 0.04 | 0.0126647446 | 47 | 922.6266479492 |
| 9 | 0.06 | 0.04 | 0.0156201366 | 99 | 2396.9099121094 |
| 9 | 0.05 | 0.06 | 0.0189898219 | 53 | 1560.0139160156 |
| 9 | 0.08 | 0.04 | 0.0193921700 | 77 | 2314.4555664062 |

(Article 1 & 2 - Top 10 list by Ending Fitness and Solution Rating – Minimisation One)

Seen in articles 1 and 2, there is an observed consistency with the mutation height and rate. All bar two examples, the mutation rate is relatively larger than the mutation height. That said, it is important to also recognise how small these arguments should be for optimal results. Top ranges for both tables do not exceed 0.11.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| %S | %MR | %MH | Ending Fitness | Plateau | Solution Fitness Rating |
| 5 | 0.04 | 0.04 | 0.0000293466 | 84 | 3.8209300041 |
| 5 | 0.04 | 0.05 | 0.0000410041 | 65 | 4.1311583519 |
| 5 | 0.04 | 0.06 | 0.0000590573 | 52 | 4.7600188255 |
| 5 | 0.04 | 0.07 | 0.0000804020 | 49 | 6.1065320969 |
| 5 | 0.04 | 0.08 | 0.0001050426 | 46 | 7.4895405769 |
| 8 | 0.05 | 0.04 | 0.0001229856 | 70 | 13.3439388275 |
| 5 | 0.04 | 0.09 | 0.0001329845 | 43 | 8.8634157181 |
| 9 | 0.04 | 0.04 | 0.0001355237 | 50 | 10.5030860901 |
| 8 | 0.04 | 0.07 | 0.0001453827 | 62 | 13.9712820053 |
| 8 | 0.05 | 0.05 | 0.0001592535 | 57 | 14.0700502396 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| %S | %MR | %MH | Ending Fitness | Plateau | Solution Fitness Rating |
| 5 | 0.04 | 0.04 | 0.0000293466 | 84 | 3.8209300041 |
| 5 | 0.04 | 0.05 | 0.0000410041 | 65 | 4.1311583519 |
| 5 | 0.04 | 0.06 | 0.0000590573 | 52 | 4.7600188255 |
| 5 | 0.04 | 0.07 | 0.0000804020 | 49 | 6.1065320969 |
| 5 | 0.04 | 0.08 | 0.0001050426 | 46 | 7.4895405769 |
| 5 | 0.04 | 0.09 | 0.0001329845 | 43 | 8.8634157181 |
| 9 | 0.04 | 0.04 | 0.0001355237 | 50 | 10.5030860901 |
| 8 | 0.05 | 0.04 | 0.0001229856 | 70 | 13.3439388275 |
| 8 | 0.04 | 0.07 | 0.0001453827 | 62 | 13.9712820053 |
| 8 | 0.05 | 0.05 | 0.0001592535 | 57 | 14.0700502396 |

(Article 3 & 4 – Top 10 list by Ending Fitness and Solution Rating – Minimisation Two)

The function dependant relationship explained by articles 1 and 2 are also mirrored by minimisation two: as shown in articles 3 and 4. Here, the relationship prefers for the mutation height to be dominant over the mutation rate. Interestingly, there is also a change in preference to the ideal tournament size. In minimisation one, there is a preference to the higher spectrum arguments. The second function works better with mid-tier tournaments.

Something else presents itself whilst reviewing articles 1 through 4: the ending fitness is a factor in deciding a solutions fitness, but just because a solution has a low fitness does not mean its arguments are efficient. The top solutions list will commonly contain similar results, but usually in different orders to better represent the arguments attributed to the ending fitness.

# Comparison

Knowing these relationships exist between the arguments and the solutions efficiency, it would be good to apply that knowledge to a new minimisation function and compare the results. Choosing the right minimisation function is inconsequential, regardless: the Trid test function was originally noted by Neumaier as a “hard global optimisation test” A black and white math equation

Description automatically generatedfunction. By modern standards, it is relatively simple. However, for the purpose of the test, it’s perfectly suited.

Given the previous results. The defined relationships suggest that the optimal range of mutation height and rate should be between 0.05 and 0.07. The tournament size is unpredictable, however, given the trade-off: tournament size for function calls, its fair to assume the tournament size would be around 6.

Neumaier A. (1996), Revisited by Adorio E. (2005).

Interestingly, this formula produces an ending fitness graph with large similarities to Minimisation Two. The key difference is that this graph typically deals with negatives. A screen shot of a graph

Description automatically generated

(Figure 16 – Ending Fitness Graph for Trid Function – Inspection Study)

Comparisons between the three different test functions show direct proof between the discussed relationships and the results given. While the wave formation of the graph changes depending on the fitness equation used, the overall effect of the parameters stay consistent. It is important to also ensure that enough experience has been gained with the specific model used. This requires trying different values for each new function proposed. Ultimately specific values may be more desirable: as proven by minimisation one, where an even tournament size would ensure a tighter range in ending fitness.

To justify a comparison, one must consider different approaches entirely. A commonly used optimisation approach is called a Hillclimb method. This method takes the premise that the climber will start at any random point on a hill. They continually take random steps in every direction until an improvement is made. In a linear environment, such as the represented Individuals genes, this is represented as the gene getting minimised or maximised.

|  |  |
| --- | --- |
| Parameter | Value |
| Tournament Size | 7 |
| Mutation Rate | 0.07 |
| Mutation Height | 0.05 |

(Article 5 – Table of test arguments used in Hillclimber v EA Comparison)

A screen shot of a black screen

Description automatically generated

(Figure 17 – Fitness graph for Hillclimber – Minimisation One)

The use of the Hillclimb method is treated with caution, but it should still be explored as a viable option in specific situations. As seen in Figure 17, in comparison to Figure 18, when facing an unoptimized Evolutionary Algorithm’s using the same minimisation function: the resulting fitness is drastically worse. A comparison cannot be made without first looking at time complexity. It took the Hillclimb 0.310258s to find a fitness below 0.65, completing 488,954 “steps” in the process. The Evolutionary algorithm averaged a time of 0.014663s, instead, reaching an end fitness of 0.01366 over 99 generations. The evolutionary algorithm is 21.18x more efficient, and 47.6x more effective with regards to Minimisation One.

A screen shot of a video player

Description automatically generated

(Figure 19 – Graph of Fitness for Hillclimber – Minimisation Two)

A black screen with white text

Description automatically generated

(Figure 20 – Graph of Fitness for Evolutionary Algorithm – Minimisation Two)

Akin to Minimisation One, Minimisation Two is also dominated by the evolutionary algorithm. It reaches a much lower final ending fitness of 0.00302619, over 34.276558; also taking 0.0110415s instead of 1.15081. This represents a 99.991% increase in effectiveness and 99.042% for efficiency.

A black and red screen

Description automatically generated

(Figure 21 – Graph of Fitness for Hillclimber – Minimisation Three – Trid Function)

A black screen with white text

Description automatically generated

(Figure 22 - Graph of Fitness for Evolutionary Algorithm – Minimisation Three – Trid Function)

Once again, mass improvements are seen regarding the efficiency of the third minimisation function seen in Figures 21 and 22. The Evolutionary Algorithm reaches a minimum of -8.98705, whilst the hill climb only reaches -7.136455. This is much better than previous tests, with a difference of only 25.8%. Time differences are still something to be desired, the hill climb took a total of 0.16996s, whilst the Evolutionary algorithm only took 0.0128104s: showing a 92.4% speed increase.

# Conclusions

The most important thing to consider when selecting a type of minimisation approach is the type of problem that needs to be solved, and how quickly it needs to be solved. In these instances, the evolutionary algorithm is a clear choice. It consistently ends at a more admirable fitness and takes significantly less time than its hill climbing counterpart.

When dealing with evolutionary algorithms, the outcome of the solution is entirely dependant on the arguments fed into it. A key established relationship is between a low mutation rate and height: resulting in a lower ending fitness. This is not to be confused with a mutation rate and height that is too low, so that it takes too many generations to reach an end. An ideal range is between 0.03 and 0.11. There is a risk with having the arguments as low as 0.03, it can often cause spikes in ending fitness by taking too many generations. Risks of low mutations are a serious threat, as it will often result in the original fitness being similar to the ending fitness.

The other key relationship that exists within the evolutionary algorithm is the Tournament selection parameters. Increasing the number of challengers increases the chance of duplication for the strongest fighters. This can be very advantageous. If the tournament selection becomes too big, the program will take too long to execute; resulting in a degraded efficiency of the entire solution.

Without extreme caution when setting arguments, there is the evident risk that the ending fitness would be extremely unpredictable. It is ideal to stick within the recommended values of 5-9 for tournaments, and between 0.04 and 0.11 for the mutation height or rate. These boundaries will often return the most efficient results.

# References

Neumaier A. (1996) Some Hard Global Optimization Test Problems. ***University of Vienna*** [online]. [Accessed 08/12/2023].

Adorio E. (2005) Multivariate Test Functions for Unconstrained Global Optimization. ***University of Philippines Diliman, Department of Mathematics*** [online]. [Accessed 08/12/2023]

# Appendix:

## External Source Code:

1. Github – Dear ImGui - <https://github.com/ocornut/imgui>

Dear ImGui is an open-source GUI library which can use a multitude of different graphics drivers. This instance has been configured using directx9 and is windows exclusive.

1. Github – ImPlot - <https://github.com/epezent/implot>

ImPlot is an open-source graph rendering library which builds on the Dear ImGui framework. It allows for multiple different types of graphs to be easily displayed with impressive zoom and detail features.

## Internal Source Code:

### Github - <https://github.com/Sp0xF8/AI-2>

### Zip – see other submissible

### main.cpp –

#include <assignment.h>

#include <graphing.h>

#include <defines.h>

#include <chrono>

#include <iostream>

int main (){

    #ifdef \_TIMER

        std::chrono::steady\_clock totalelapsed;

        auto start = std::chrono::high\_resolution\_clock::now();

        for (int i = 0; i < 100; i++){

    #endif

    #ifdef RUN\_GENETIC\_ALGORITHM

            Assignment::runAssignment();

    #endif

    #ifdef RUN\_HILL\_CLIMBER

            Assignment::runHillClimber();

    #endif

    #ifdef \_TIMER

        }

        auto end = std::chrono::high\_resolution\_clock::now();

        std::chrono::duration<double> elapsed = end - start;

        float average = elapsed.count() / 100;

        std::cout << "Time taken by program is : " << average << " seconds" << std::endl;

    #endif

    gui::graphing();

*//ask user to close program*

    std::cout << "Press any key to close program...";

    std::cin.get();

    return 0;

}

### Assignment.h –

#pragma once

namespace Assignment {

    bool runAssignment();

    bool runHillClimber();

};

### Graphing.h –

pragma once

#include <d3d9.h>

namespace gui {

    constexpr int WIDTH = 1900;

    constexpr int HEIGHT = 1040;

    inline bool exit = true;

    inline HWND window = nullptr;

    inline WNDCLASSEXA windowClass = {};

    inline POINTS position = {};

    inline PDIRECT3D9 d3d = nullptr;

    inline LPDIRECT3DDEVICE9 device = nullptr;

    inline D3DPRESENT\_PARAMETERS presentParameters = {};

    void CreateHWindow(

        const char\* windowName,

        const char\* className) noexcept;

    void DestroyHWindow() noexcept;

    bool CreateDevice() noexcept;

    void ResetDevice() noexcept;

    void DestroyDevice() noexcept;

    void CreateImGui() noexcept;

    void DestroyImGui() noexcept;

    void BeginRender() noexcept;

    void EndRender() noexcept;

    void Menu() noexcept;

    void Theme() noexcept;

    void Render() noexcept;

    bool graphing() noexcept;

}

### Defines.h

#pragma once

*// #define RUN\_HILL\_CLIMBER*

#define RUN\_GENETIC\_ALGORITHM

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   SETTINGS FOR FINDING THE BEST OR WORST INDIVIDUAL*

*//     - FIND\_BEST : FINDS THE BEST INDIVIDUAL*

*//*

*//     OTHERWISE THE PROGRAM WILL FIND THE WORST INDIVIDUAL BY DEFAULT*

*//*

*///////////////////*

*// #define FIND\_BEST*

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   SELECT A SPECIFIC TEST FUNCTION*

*//     ONLY ONE OF THE FOLLOWING CAN BE DEFINED AT A TIME*

*//*

*//     - SIMPLE\_TEST : SIMPLE TEST FUNCTION*

*//     - ADVANCED\_TEST : ADVANCED TEST FUNCTION [MINIMISATION AND MAXIMISATION]*

*//*

*///////////////////*

#ifdef FIND\_BEST

    #define SIMPLE\_TEST

#else

*// #define ADVANCED\_TEST*

*// #define COMPLEX\_TEST*

    #define TRID\_FUNCTION

#endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//  ONLY COMMENT ONE OF THE FOLLOWING:*

*//      - ELITEISM - HEAVY\_ELITEISM - SIMPLE\_PASSDOWN*

*//          OTHERWISE THE PROGRAM WILL CRASH*

*//*

*//      ELITEISM : CHECKS IF ANY PROGRESSION WAS MADE FROM THE PREVIOUS GENERATION*

*//      HEAVY\_ELITEISM : FORCES THE BEST OF ALL 4 POPULATIONS INTO THE NEXT GENERATION*

*//      SIMPLE\_PASSDOWN : FORCES THE MUTATED POPULATION ONTO THE NEXT GENERATION*

*//*

*///////////////////*

#define ELITEISM

*// #define HEAVY\_ELITEISM*

*// #define SIMPLE\_PASSDOWN*

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//    RANDOM\_SEED : SEED FOR RANDOM NUMBER GENERATOR -*

*//      RAMDOM SEED IS OUTPUTTED TO THE CONSOLE WHEN THE SEED == 0*

*//*

*///////////////////*

#define RANDOM\_SEED 1699118515

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   POPULATION\_SIZE : NUMBER OF INDIVIDUALS IN A POPULATION*

*//   NUMBER\_OF\_GENES : NUMBER OF GENES IN AN INDIVIDUAL*

*//   NUMBER\_OF\_GENERATIONS : NUMBER OF GENERATIONS TO RUN THE PROGRAM FOR*

*//*

*///////////////////*

#define POPULATION\_SIZE 50

#define NUMBER\_OF\_GENES 10

#define NUMBER\_OF\_GENERATIONS 100

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   TOURNAMENT\_SIZE : NUMBER OF INDIVIDUALS IN A GLADIATOR TOURNAMENT*

*//*

*///////////////////*

#ifdef FIND\_BEST

    #define TOURNAMENT\_SIZE 3

#else

    #define TOURNAMENT\_SIZE 7

#endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   MUTATION\_RATE : THE PROBABILITY OF A MUTATION*

*//   MUTATION\_HEIGHT : THE UPPER AND LOWER BOUND OF THE MUTATION*

*//*

*//      DIFFERENT SETTINGS ARE USED FOR FINDING BEST OR WORST*

*//*

*///////////////////*

#ifdef FIND\_BEST

    #define MUTATION\_RATE 0.28

    #define MUTATION\_HEIGHT 0.02

#else

    #define MUTATION\_RATE 0.07

    #define MUTATION\_HEIGHT 0.05

#endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   PLATO\_HEIGHT : THE HEIGHT OF WHAT IS CONSIDERED A PLATO*

*//*

*///////////////////*

#define PLATO\_HEIGHT 0.005

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   DIFFERENT DEBUGGING SETTINGS*

*//*

*//   \_DEBUG\_GENES : PRINTS THE GENES OF AN INDIVIDUAL*

*//   \_DEBUG\_FITNESS : PRINTS THE FITNESS OF AN INDIVIDUAL*

*//   \_DEBUG\_INDIVIDUAL : PRINTS THE GENES AND FITNESS OF AN INDIVIDUAL*

*//   \_DEBUG\_GENERATION : PRINTS INFORMATION ABOUT THE CURRENT GENERATIONS PROGRESSION*

*//*

*//   \_DEBUG\_POPULATION : PRINTS THE GENES AND FITNESS OF A POPULATION*

*//   \_DEBUG\_GLADIATOR : PRINTS THE GENES AND FITNESS OF A GLADIATOR*

*//   \_DEBUG\_CROSSPOINT : PRINTS THE GENES AND FITNESS OF A CROSSPOINT*

*//   \_DEBUG\_MUTATION : PRINTS THE GENES AND FITNESS OF A MUTATION*

*//*

*///////////////////*

*// #define \_DEBUG\_GENES*

*// #define \_DEBUG\_FITNESS*

*// #define \_DEBUG\_INDIVIDUAL*

*// #define \_DEBUG\_GENERATION*

*// #define \_DEBUG\_POPULATION*

*// #define \_DEBUG\_GLADIATOR*

*// #define \_DEBUG\_CROSSPOINT*

*// #define \_DEBUG\_MUTATION*

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   DIFFERENT GRAPHING SETTINGS*

*//*

*//   \_PLOT\_GRAPHS : THIS IS REQUIRED FOR ANY OF THE FOLLOWING TO WORK*

*//        IF THIS IS NOT DEFINED, THE PROGRAM WILL NOT CALL THE FUNCTIONS TO PLOT DATA*

*//*

*//   \_GRAPH\_POPULATION : PLOTS THE FITNESS OF THE POPULATION*

*//   \_GRAPH\_GLADIATOR : PLOTS THE FITNESS OF THE GLADIATOR*

*//   \_GRAPH\_CROSSPOINT : PLOTS THE FITNESS OF THE CROSSPOINT*

*//   \_GRAPH\_MUTATION : PLOTS THE FITNESS OF THE MUTATION*

*//*

*///////////////////*

#define \_PLOT\_GRAPHS

#ifdef \_PLOT\_GRAPHS

    #define \_GRAPH\_POPULATION

    #define \_GRAPH\_GLADIATOR

    #define \_GRAPH\_CROSSPOINT

    #define \_GRAPH\_MUTATION

    #define \_GRAPH\_GENERATION\_HISTORY

#endif

*// #define \_GRAPH\_FITNESS*

*// #define \_GRAPH\_FITNESS\_AVERAGE*

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//   TOGGLE META AI*

*//*

*//   \_META\_AI : TOGGLES META AI*

*//*

*///////////////////*

*// #define \_META\_AI*

#define \_TIMER

#define NUMBER\_OF\_RESULTS 10

### Assignment.cpp –

#include <individual.h>

#include <helper.h>

#include <assignment.h>

#include <iostream>

#ifdef \_PLOT\_GRAPHS

#else

    float fitnesses[4];

#endif

int g = 0;

#ifdef \_META\_AI

    #include <vector>

    float mutation\_rate = 0.0;

    float mutation\_height = 0.0;

    int tournament\_size = 2;

    float best\_mutation\_rate = 0.0;

    float best\_mutation\_height = 0.0;

    int best\_tournament\_size = 2;

    #ifdef FIND\_BEST

        float ending\_fitness\_height = 0.0;

    #else

        float ending\_fitness\_height = 1.0;

    #endif

#endif

void\* population[POPULATION\_SIZE];

void\* gladiators[POPULATION\_SIZE];

void\* crosspoints[POPULATION\_SIZE];

void\* mutations[POPULATION\_SIZE];

int tempPlato = 0;

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 SIMPLE FUNCTION TO GENERATE A POPULATION*

*//      TAKES ARGUMENTS:*

*//          #NONE - USES GLOBAL VARIABLES EXISTING INSIDE THIS CPP FILE*

*//*

*//*

*//      CYCLES THROUGH EACH INDIVIDUAL IN THE POPULATION*

*//          GENERATES GENES FOR EACH INDIVIDUAL*

*//          CALCULATES THE FITNESS OF EACH INDIVIDUAL*

*//          ADDS THE INDIVIDUAL TO THE POPULATION*

*//*

*//      RETURNS:*

*//          true : IF THE POPULATION WAS GENERATED*

*//          false : IF THE POPULATION WAS NOT GENERATED*

*//*

*///////////////////*

bool GeneratePopulatin(){

    for (int i = 0; i < POPULATION\_SIZE; i++) {

        Individual \*individual = new Individual();

        Helper::generateGenes(individual); *// Populate initial population*

        population[i] = individual;

        Helper::calculateFitness(individual); *// Calculate fitness of each individual*

    }

    return true;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          GLADIATOR SELECTION FUNCTION*

*//      TAKES ARGUMENTS:*

*//          #NONE - USES GLOBAL VARIABLES EXISTING INSIDE THIS CPP FILE*

*//*

*//*

*//      CYCLES THROUGH EACH INDIVIDUAL IN THE POPULATION*

*//          GETS x AMOUNT OF RANDOM INDIVIDUALS FROM THE POPULATION*

*//          FINDS THE BEST INDIVIDUAL FROM THE RANDOM INDIVIDUALS*

*//          CLONES THE BEST INDIVIDUAL TO THE GLADIATORS POPULATION*

*//*

*//      RETURNS:*

*//          true : IF THE POPULATION WAS GLADIATOR SELECTED*

*//          false : IF THE POPULATION WAS NOT GLADIATOR SELECTED*

*//*

*///////////////////*

bool GladiatorSelection(){

    #ifdef \_META\_AI

        for(int i = 0; i < POPULATION\_SIZE; i++){

            std::vector<Individual> selection;

            selection.push\_back(\*(Individual\*)population[i]);

            for(int j = 1; j < tournament\_size; j++){

                int random\_index = rand() % POPULATION\_SIZE;

                selection.push\_back(\*(Individual\*)population[random\_index]);

            }

            Individual best = selection[0];

            #ifdef FIND\_BEST

                for(int j = 1; j < tournament\_size; j++){

                    if(selection[j].getFitness() > best.getFitness()){

                        best = selection[j];

                    }

                }

            #else

                for(int j = 1; j < tournament\_size; j++){

                    if(selection[j].getFitness() < best.getFitness()){

                        best = selection[j];

                    }

                }

            #endif

            Individual \*gladiator = new Individual();

            gladiator->setGenes(best.getGenes());

            gladiator->setFitness(best.getFitness());

            gladiators[i] = gladiator;

        }

    #else

        for (int i = 0; i < POPULATION\_SIZE; i++) {

            Individual selection[TOURNAMENT\_SIZE];

            selection[0] = \*(Individual\*)population[i];

            for (int j = 1; j < TOURNAMENT\_SIZE; j++) {

                int random\_index = rand() % POPULATION\_SIZE;

                selection[j] = \*(Individual\*)population[random\_index];

            }

            #ifdef FIND\_BEST

*//find best individual from selection*

                Individual best = selection[0];

                for (int j = 1; j < TOURNAMENT\_SIZE; j++) {

                    if(selection[j].getFitness() > best.getFitness()){

                        best = selection[j];

                    }

                }

            #else

*//find worst individual from selection*

                Individual best = selection[0];

                for (int j = 1; j < TOURNAMENT\_SIZE; j++) {

                    if(selection[j].getFitness() < best.getFitness()){

                        best = selection[j];

                    }

                }

            #endif

*//clone best individual to gladiators*

            Individual \*gladiator = new Individual();

            gladiator->setGenes(best.getGenes());

            gladiator->setFitness(best.getFitness());

            gladiators[i] = gladiator;

        }

    #endif

    return true;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          Crosspoint Function*

*//      TAKES ARGUMENTS:*

*//          #NONE - USES GLOBAL VARIABLES EXISTING INSIDE THIS CPP FILE*

*//*

*//*

*//      CYCLES THROUGH EACH INDIVIDUAL IN THE POPULATION*

*//          GETS 2  RANDOM INDIVIDUALS FROM THE POPULATION*

*//          GETS A RANDOM CROSSPOINT*

*//              SETS THE GENES OF THE NEW INDIVIDUALS TO THE GENES OF THE RANDOM INDIVIDUALS*

*//              SWAPS THE GENES OF THE NEW INDIVIDUALS AT THE CROSSPOINT*

*//              CALCULATES THE FITNESS OF THE NEW INDIVIDUALS*

*//              ADDS THE NEW INDIVIDUALS TO THE CROSSPOINT POPULATION*

*//*

*//      RETURNS:*

*//          true : IF THE POPULATION WAS CROSSPOINTED*

*//          false : IF THE POPULATION WAS NOT CROSSPOINTED*

*//*

*///////////////////*

bool Crosspoint(){

        for (int i = 0; i < POPULATION\_SIZE; i += 2) {

            Individual \*individual1 = new Individual();

            Individual \*individual2 = new Individual();

*//get 2 random gladiators*

            int random\_index1 = rand() % POPULATION\_SIZE;

            int random\_index2 = rand() % POPULATION\_SIZE;

*//get genes from gladiators*

            float \*genes1 = ((Individual\*)gladiators[random\_index1])->getGenes();

            float \*genes2 = ((Individual\*)gladiators[random\_index2])->getGenes();

*//get random crosspoint*

            int random\_crosspoint = rand() % NUMBER\_OF\_GENES;

*//set genes for new individuals*

            float temp\_gene;

            for (int j = random\_crosspoint; j < NUMBER\_OF\_GENES; j++) {

                temp\_gene = genes1[j];

                genes1[j] = genes2[j];

                genes2[j] = temp\_gene;

            }

            individual1->setGenes(genes1);

            individual2->setGenes(genes2);

*//calculate fitness*

            Helper::calculateFitness(individual1);

            Helper::calculateFitness(individual2);

*//add to crosspoint*

            crosspoints[i] = individual1;

            crosspoints[i + 1] = individual2;

        }

        return true;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          Mutation Function*

*//      TAKES ARGUMENTS:*

*//          #NONE - USES GLOBAL VARIABLES EXISTING INSIDE THIS CPP FILE*

*//*

*//*

*//      CYCLES THROUGH EACH INDIVIDUAL IN THE POPULATION*

*//          AND CYCLES THROUGH EACH GENE IN THE INDIVIDUAL*

*//          GENERATES A RANDOM FLOAT BETWEEN 0 AND 1*

*//          IF THE RANDOM FLOAT IS LESS THAN THE MUTATION RATE*

*//              THEN GENERATE A RANDOM FLOAT BETWEEN 0 AND THE MUTATION HEIGHT*

*//              AND ADD IT TO THE GENE*

*//*

*//      RETURNS:*

*//          true : IF THE POPULATION WAS MUTATED*

*//          false : IF THE POPULATION WAS NOT MUTATED*

*//*

*///////////////////*

bool Mutation(){

    #ifdef \_META\_AI

        for(int i = 0; i < POPULATION\_SIZE; i++){

            float\* genes = ((Individual\*)crosspoints[i])->getGenes();

            for (int j = 0; j < NUMBER\_OF\_GENES; j++){

                float random = (float)rand() / RAND\_MAX;

                if(random < mutation\_rate){

                    float alteration = (float)rand() / RAND\_MAX \* mutation\_height;

                    float negative = (float)rand() / RAND\_MAX;

                    #ifdef \_DEBUG\_MUTATION

                        printf("Mutation!\n");

                        printf("Gene: %f + %f\n", genes[j], alteration);

                    #endif

                    if (negative < 0.5) {

                        genes[j] += alteration;

                    } else {

                        genes[j] -= alteration;

                    }

                    if(genes[j] > 1)

                        genes[j] = 1;

                    if(genes[j] < 0)

                        genes[j] = 0;

                }

            }

            Individual \*individual = new Individual();

            individual->setGenes(genes);

            Helper::calculateFitness(individual);

            mutations[i] = individual;

        }

    #else

        for (int i = 0; i < POPULATION\_SIZE; i++) {

            float\* genes = ((Individual\*)crosspoints[i])->getGenes();

            for (int j = 0; j < NUMBER\_OF\_GENES; j++) {

                float random = (float)rand() / RAND\_MAX;

                if(random < MUTATION\_RATE){

                    float alteration = (float)rand() / RAND\_MAX \* MUTATION\_HEIGHT;

                    float negative = (float)rand() / RAND\_MAX;

                    #ifdef \_DEBUG\_MUTATION

                        printf("Mutation!\n");

                        printf("Gene: %f + %f\n", genes[j], alteration);

                    #endif

                    if (negative < 0.5) {

                        genes[j] += alteration;

                    } else {

                        genes[j] -= alteration;

                    }

                    if(genes[j] > 1)

                        genes[j] = 1;

                    if(genes[j] < 0)

                        genes[j] = 0;

                }

            }

            Individual \*individual = new Individual();

            individual->setGenes(genes);

            Helper::calculateFitness(individual);

            mutations[i] = individual;

        }

    #endif

    return true;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                    REPLACE POPULATION WITH BEST POPULATION*

*//      TAKES ARGUMENTS:*

*//          best\_population : THE POPULATION TO REPLACE THE CURRENT POPULATION WITH*

*//*

*//*

*//      DELETES THE CURRENT POPULATION AND REPLACES IT WITH THE BEST POPULATION*

*//          IF THE BEST POPULATION IS THE SAME AS THE CURRENT POPULATION, IT WILL DELETE OTHER POPULATIONS*

*//          OTHERWISE, IT WILL DELETE THE CURRENT POPULATION AND REPLACE IT WITH THE BEST POPULATION*

*//*

*//      RETURNS:*

*//          true : IF THE POPULATION WAS REPLACED*

*//          false : IF THE POPULATION WAS NOT REPLACED*

*//*

*///////////////////*

bool ReplacePopulation(void\* best\_population[]){

    #ifdef \_GRAPH\_GENERATION\_HISTORY

        Helper::CopyPopulation(population, Helper::generation\_history[g].population);

        Helper::CopyPopulation(gladiators, Helper::generation\_history[g].gladiators);

        Helper::CopyPopulation(crosspoints, Helper::generation\_history[g].crosspoints);

        Helper::CopyPopulation(mutations, Helper::generation\_history[g].mutations);

    #endif

    void\* temp\_population[POPULATION\_SIZE];

    if(best\_population == population){

*// printf("Replacing population with itself\n");*

    } else {

        Helper::CopyPopulation(best\_population, temp\_population);

        Helper::ClearPopulation(population);

        Helper::CopyPopulation(temp\_population, population);

*// ClearPopulation(temp\_population);*

    }

    Helper::ClearPopulation(gladiators);

    Helper::ClearPopulation(crosspoints);

    Helper::ClearPopulation(mutations);

    return true;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO FIGURE OUT IF THE GENERATIONS HAVE PLATOED*

*//      TAKES ARGUMENTS:*

*//          NONE*

*//*

*//*

*//      CHECKS IF THE FITNESS OF THE CURRENT GENERATION IS WITHIN BOUNDS OF A GEBERATION 10 GENERATIONS AGO*

*//          IF IT IS THEN THE GENERATIONS HAVE PLATOED*

*//*

*//      RETURNS:*

*//          true : IF THE GENERATIONS HAVE PLATOED*

*//          false : IF THE GENERATIONS HAVE NOT PLATOED*

*//*

*///////////////////*

bool CheckPlato(){

    if(g > 10){

        if( Helper::average\_fitnesses[g] < Helper::average\_fitnesses[g - 10] + PLATO\_HEIGHT

                                            &&

            Helper::average\_fitnesses[g] > Helper::average\_fitnesses[g - 10] - PLATO\_HEIGHT

            ){

            return true;

        }

    }

    return false;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 THIS IS THE PART OF THE PROGRAM WHICH IS CALLED FROM MAIN*

*//      TAKES ARGUMENTS:*

*//          #NONE*

*//*

*//*

*//      THIS FUNCTION RUNS THE ENTIRE PROGRAM. IT GENERATES THE INITIAL POPULATION, THEN RUNS THE GENERATIONS LOOP*

*//*

*//                                                   THIS FUNCTION IS CALLED EXTERNALLY, INSTEAD OF MAIN TO ALLOW FOR*

*//                                                   EASY TESTING OF THE PROGRAM AND BETTER READABILITY*

*//*

*//      RETURNS:*

*//          true : IF THE PROGRAM WAS RUN SUCCESSFULLY*

*//          false : IF THE PROGRAM WAS NOT RUN SUCCESSFULLY*

*//*

*///////////////////*

bool Assignment::runAssignment(){

    #ifdef \_META\_AI

    for (tournament\_size = 3; tournament\_size < 10; tournament\_size++) {

        for ( mutation\_rate = 0.04; mutation\_rate < 0.25; mutation\_rate += 0.01) {

            for ( mutation\_height = 0.04; mutation\_height < 0.2; mutation\_height += 0.01) {

*// printf("Mutation rate: %f\n", mutation\_rate);*

*// printf("Mutation height: %f\n", mutation\_height);*

*// printf("Tournament size: %d\n", tournament\_size);*

    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                              DATA VARIATION*

*//*

*///////////////////*

                #ifdef RANDOM\_SEED

                    if(RANDOM\_SEED != 0){

                        srand(RANDOM\_SEED);

                    }else{

                        printf("Seed is 0, using time as seed\n");

                        int seed = time(NULL);

                        printf("Random seed: %d\n", seed);

                        srand(seed);

                    }

                #else

                    int seed = time(NULL);

                    printf("Random seed: %d\n", seed);

                    srand(seed);

                #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          GENERATE INITIAL POPULATION*

*//*

*///////////////////*

                if(!GeneratePopulatin()){

                    printf("Error generating population\n");

                    return false;

                }

                #ifdef \_DEBUG\_POPULATION

                    printf("\n\nInitial population\n\n");

                    Helper::printPopulation(population);

                #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                                  GENERATIONS*

*//*

*///////////////////*

                for (g = 0; g < NUMBER\_OF\_GENERATIONS; g++) {

*/\* code \*/*

                    #ifdef \_DEBUG\_GENERATION

                        printf("\n\nGeneration %d\n", g);

                    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          GLADIATOR SELECTION*

*//*

*///////////////////*

                    if(!GladiatorSelection()){

                        printf("Error selecting gladiators\n");

                        return false;

                    }

                    #ifdef \_DEBUG\_GLADIATOR

                        printf("\n\n Gladiators\n\n");

                        Helper::printPopulation(gladiators);

                    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                              CROSSPOINT*

*//*

*///////////////////*

                    if(!Crosspoint()){

                        printf("Error crosspointing\n");

                        return false;

                    }

                    #ifdef \_DEBUG\_CROSSPOINT

                        printf("\n\n Crosspoints\n\n");

                        Helper::printPopulation(crosspoints);

                    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                                MUTATION*

*//*

*///////////////////*

                    if(!Mutation()){

                        printf("Error mutating\n");

                        return false;

                    }

                    #ifdef \_DEBUG\_MUTATION

                        printf("\n\n Mutations\n\n");

                        Helper::printPopulation(mutations);

                    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                              CALCULATE FITNESS FOR EACH POPULATION AND STORE FOR GRAPHING*

*//                                              IF GRAPHING IS ENABLED*

*//                              OTHERWISE, STORE FITNESS FOR EACH POPULATION TEMPORARILY*

*//*

*///////////////////*

                    #ifdef \_PLOT\_GRAPHS

                        #ifdef FIND\_BEST

                            Helper::generations[g].population.best = Helper::getPopulationHeight(population);

                            Helper::generations[g].gladiator.best = Helper::getPopulationHeight(gladiators);

                            Helper::generations[g].crosspoint.best = Helper::getPopulationHeight(crosspoints);

                            Helper::generations[g].mutation.best = Helper::getPopulationHeight(mutations);

                        #else

                            Helper::generations[g].population.worst = Helper::getPopulationHeight(population);

                            Helper::generations[g].gladiator.worst = Helper::getPopulationHeight(gladiators);

                            Helper::generations[g].crosspoint.worst = Helper::getPopulationHeight(crosspoints);

                            Helper::generations[g].mutation.worst = Helper::getPopulationHeight(mutations);

                        #endif

                        Helper::generations[g].population.total = Helper::getPopulationFitness(population);

                        Helper::generations[g].population.average = Helper::generations[g].population.total / POPULATION\_SIZE;

                        Helper::generations[g].gladiator.total = Helper::getPopulationFitness(gladiators);

                        Helper::generations[g].gladiator.average = Helper::generations[g].gladiator.total / POPULATION\_SIZE;

                        Helper::generations[g].crosspoint.total = Helper::getPopulationFitness(crosspoints);

                        Helper::generations[g].crosspoint.average = Helper::generations[g].crosspoint.total / POPULATION\_SIZE;

                        Helper::generations[g].mutation.total = Helper::getPopulationFitness(mutations);

                        Helper::generations[g].mutation.average = Helper::generations[g].mutation.total / POPULATION\_SIZE;

                    #else

                        fitnesses[0] = Helper::getPopulationFitness(population);

                        fitnesses[1] = Helper::getPopulationFitness(gladiators);

                        fitnesses[2] = Helper::getPopulationFitness(crosspoints);

                        fitnesses[3] = Helper::getPopulationFitness(mutations);

                    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                      PRINT CURRENT GENERATION'S POPULATIONS' FITNESSES*

*//*

*///////////////////*

                    #ifdef \_PLOT\_GRAPHS

                    #else

                        #ifdef \_DEBUG\_GENERATION

                            printf("\n\nFitnesses:\n");

                            printf("Population: %f\n", fitnesses[0]);

                            printf("Gladiators: %f\n", fitnesses[1]);

                            printf("Crosspoint: %f\n", fitnesses[2]);

                            printf("Mutations: %f\n", fitnesses[3]);

                        #endif

                    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          REPLACE POPULATION WITH HIGHEST FITNESS*

*//*

*///////////////////*

                    #ifdef HEAVY\_ELITEISM

                        #ifdef \_PLOT\_GRAPHS

                            #ifdef FIND\_BEST

                                if (Helper::generations[g].population.best > Helper::generations[g].gladiator.best) {

                                    if(Helper::generations[g].population.best > Helper::generations[g].crosspoint.best)

                                        if(Helper::generations[g].population.best > Helper::generations[g].mutation.best)

                                            ReplacePopulation(population);

                                        else

                                            ReplacePopulation(mutations);

                                    else

                                        if(Helper::generations[g].crosspoint.best > Helper::generations[g].mutation.best)

                                            ReplacePopulation(crosspoints);

                                        else

                                            ReplacePopulation(mutations);

                                }else{

                                    if(Helper::generations[g].gladiator.best > Helper::generations[g].crosspoint.best)

                                        if(Helper::generations[g].gladiator.best > Helper::generations[g].mutation.best)

                                            ReplacePopulation(gladiators);

                                        else

                                            ReplacePopulation(mutations);

                                    else

                                        if(Helper::generations[g].crosspoint.best > Helper::generations[g].mutation.best)

                                            ReplacePopulation(crosspoints);

                                        else

                                            ReplacePopulation(mutations);

                                }

                            #else

                                if (Helper::generations[g].population.worst < Helper::generations[g].gladiator.worst) {

                                    if(Helper::generations[g].population.worst < Helper::generations[g].crosspoint.worst)

                                        if(Helper::generations[g].population.worst < Helper::generations[g].mutation.worst)

                                            ReplacePopulation(population);

                                        else

                                            ReplacePopulation(mutations);

                                    else

                                        if(Helper::generations[g].crosspoint.worst < Helper::generations[g].mutation.worst)

                                            ReplacePopulation(crosspoints);

                                        else

                                            ReplacePopulation(mutations);

                                }else{

                                    if(Helper::generations[g].gladiator.worst < Helper::generations[g].crosspoint.worst)

                                        if(Helper::generations[g].gladiator.worst < Helper::generations[g].mutation.worst)

                                            ReplacePopulation(gladiators);

                                        else

                                            ReplacePopulation(mutations);

                                    else

                                        if(Helper::generations[g].crosspoint.worst < Helper::generations[g].mutation.worst)

                                            ReplacePopulation(crosspoints);

                                        else

                                            ReplacePopulation(mutations);

                                }

                            #endif

                        #else

                            #ifdef FIND\_BEST

                                if(fitnesses[0] > fitnesses[1]){

                                    if(fitnesses[0] > fitnesses[2])

                                        if(fitnesses[0] > fitnesses[3])

                                            ReplacePopulation(population);

                                        else

                                            ReplacePopulation(mutations);

                                    else

                                        if(fitnesses[2] > fitnesses[3])

                                            ReplacePopulation(crosspoints);

                                        else

                                            ReplacePopulation(mutations);

                                } else {

                                    if(fitnesses[1] > fitnesses[2])

                                        if(fitnesses[1] > fitnesses[3])

                                            ReplacePopulation(gladiators);

                                        else

                                            ReplacePopulation(mutations);

                                    else

                                        if(fitnesses[2] > fitnesses[3])

                                            ReplacePopulation(crosspoints);

                                        else

                                            ReplacePopulation(mutations);

                                }

                            #else

                                if(fitnesses[0] < fitnesses[1]){

                                    if(fitnesses[0] < fitnesses[2])

                                        if(fitnesses[0] < fitnesses[3])

                                            ReplacePopulation(population);

                                        else

                                            ReplacePopulation(mutations);

                                    else

                                        if(fitnesses[2] < fitnesses[3])

                                            ReplacePopulation(crosspoints);

                                        else

                                            ReplacePopulation(mutations);

                                } else {

                                    if(fitnesses[1] < fitnesses[2])

                                        if(fitnesses[1] < fitnesses[3])

                                            ReplacePopulation(gladiators);

                                        else

                                            ReplacePopulation(mutations);

                                    else

                                        if(fitnesses[2] < fitnesses[3])

                                            ReplacePopulation(crosspoints);

                                        else

                                            ReplacePopulation(mutations);

                                }

                            #endif

                        #endif

                    #endif

                    #ifdef ELITEISM

                        #ifdef \_PLOT\_GRAPHS

                            #ifdef FIND\_BEST

                                if (Helper::generations[g].population.best > Helper::generations[g].mutation.best) {

                                    ReplacePopulation(population);

                                }else{

                                    ReplacePopulation(mutations);

                                }

                            #else

                                if (Helper::generations[g].population.worst < Helper::generations[g].mutation.worst) {

                                    ReplacePopulation(population);

                                }else{

                                    ReplacePopulation(mutations);

                                }

                            #endif

                        #else

                            #ifdef FIND\_BEST

                                if(fitnesses[0] > fitnesses[3]){

                                    ReplacePopulation(population);

                                }else{

                                    ReplacePopulation(mutations);

                                }

                            #else

                                if(fitnesses[0] < fitnesses[3]){

                                    ReplacePopulation(population);

                                }else{

                                    ReplacePopulation(mutations);

                                }

                            #endif

                        #endif

                    #endif

                    #ifdef SIMPLE\_PASSDOWN

                        ReplacePopulation(mutations);

                    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          CALCULATE GENERATIONAL FITNESS AND HEIGHT*

*//*

*///////////////////*

                    #ifdef FIND\_BEST

                        Helper::best\_fitnesses[g] = Helper::getPopulationHeight(population);

                    #else

                        Helper::worst\_fitnesses[g] = Helper::getPopulationHeight(population);

                    #endif

                    Helper::average\_fitnesses[g] = Helper::getPopulationFitness(population) / POPULATION\_SIZE;

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          PRINT GENERATIONAL FITNESS AND HEIGHT*

*//*

*///////////////////*

                    #ifdef \_DEBUG\_GENERATION

                        printf("\n\nGeneration %d\n", g);

                        #ifdef FIND\_BEST

                            printf("Best fitness: %f\n", Helper::best\_fitnesses[g]);

                        #else

                            printf("Worst fitness: %f\n", Helper::worst\_fitnesses[g]);

                        #endif

                        printf("Average fitness: %f\n", Helper::average\_fitnesses[g]);

                    #endif

                    #ifdef \_META\_AI

                        #ifdef PLATO\_HEIGHT

                            if(tempPlato == 0){

                                if(CheckPlato()){

                                    tempPlato = g - 5;

                                }

                            }

                        #endif

                    #endif

                }

                    #ifdef RUN\_GENETIC\_ALGORITHM

                    #endif

                #ifdef \_META\_AI

*// printf("Average fitness: %f\n", Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1]);*

                    MetaData current\_data;

                    current\_data.generation = Helper::meta\_data.size();

                    current\_data.mutation\_rate = mutation\_rate;

                    current\_data.mutation\_height = mutation\_height;

                    current\_data.tournament\_size = (float)tournament\_size;

                    current\_data.ending\_fitness\_height = Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1];

                    #ifdef PLATO\_HEIGHT

                        if (tempPlato != 0){

                            current\_data.plato\_confirmed = tempPlato;

                            tempPlato = 0;

                        }else {

                            current\_data.plato\_confirmed = NUMBER\_OF\_GENERATIONS - 1;

                        }

                    #endif

                    current\_data.solution\_fitness = Helper::GetSolutionFitness(current\_data);

                    Helper::meta\_data.emplace\_back(current\_data);

                #ifdef FIND\_BEST

                    if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] > ending\_fitness\_height){

                        if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] < 10.0){

                            ending\_fitness\_height = Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1];

                            best\_mutation\_height = mutation\_height;

                            best\_mutation\_rate = mutation\_rate;

                            best\_tournament\_size = tournament\_size;

                        }

                    }

                #else

                    if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] < ending\_fitness\_height){

                        ending\_fitness\_height = Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1];

                        best\_mutation\_height = mutation\_height;

                        best\_mutation\_rate = mutation\_rate;

                        best\_tournament\_size = tournament\_size;

                    }

                #endif

            }

            #ifdef FIND\_BEST

                if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] > ending\_fitness\_height){

                    if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] < 10.0){

                        ending\_fitness\_height = Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1];

                        best\_mutation\_height = mutation\_height;

                        best\_mutation\_rate = mutation\_rate;

                        best\_tournament\_size = tournament\_size;

                    }

                }

            #else

                if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] < ending\_fitness\_height){

                    ending\_fitness\_height = Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1];

                    best\_mutation\_height = mutation\_height;

                    best\_mutation\_rate = mutation\_rate;

                    best\_tournament\_size = tournament\_size;

                }

            #endif

        }

        #ifdef FIND\_BEST

            if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] > ending\_fitness\_height){

                if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] < 10.0){

                    ending\_fitness\_height = Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1];

                    best\_mutation\_height = mutation\_height;

                    best\_mutation\_rate = mutation\_rate;

                    best\_tournament\_size = tournament\_size;

                }

            }

        #else

            if(Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1] < ending\_fitness\_height){

                ending\_fitness\_height = Helper::average\_fitnesses[NUMBER\_OF\_GENERATIONS - 1];

                best\_mutation\_height = mutation\_height;

                best\_mutation\_rate = mutation\_rate;

                best\_tournament\_size = tournament\_size;

            }

        #endif

    }

#endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          PRINT RESULTS*

*//*

*///////////////////*

    #ifdef \_META\_AI

        printf("\n\nBest mutation rate: %f\n", best\_mutation\_rate);

        printf("Best mutation height: %f\n", best\_mutation\_height);

        printf("Best tournament size: %d\n", best\_tournament\_size);

        printf("Ending fitness height: %f\n", ending\_fitness\_height);

    #endif

    #ifdef \_DEBUG\_FITNESS

        #ifdef FIND\_BEST

            printf("\n\nBest fitnesses:\n");

            for (int i = 0; i < NUMBER\_OF\_GENERATIONS; i++) {

                printf("%d:  B:[%f]  A: [%f]\n", i, Helper::best\_fitnesses[i], Helper::average\_fitnesses[i]);

            }

        #else

            printf("\n\nWorst fitnesses:\n");

            for (int i = 0; i < NUMBER\_OF\_GENERATIONS; i++) {

                printf("%d:  W:[%f]  A: [%f]\n", i, Helper::worst\_fitnesses[i], Helper::average\_fitnesses[i]);

            }

        #endif

    #endif

    return true;

}

bool Assignment::runHillClimber(){

    #ifdef RANDOM\_SEED

        if(RANDOM\_SEED != 0){

            srand(RANDOM\_SEED);

        }else{

            printf("Seed is 0, using time as seed\n");

            int seed = time(NULL);

            printf("Random seed: %d\n", seed);

            srand(seed);

        }

    #else

        int seed = time(NULL);

        printf("Random seed: %d\n", seed);

        srand(seed);

    #endif

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                          GENERATE INITIAL POPULATION*

*//*

*///////////////////*

    Individual \*HillClimber = new Individual();

    Helper::generateGenes(HillClimber); *// Populate initial population*

    Helper::calculateFitness(HillClimber); *// Calculate fitness of each individual*

    Helper::fitnesses.emplace\_back(HillClimber->getFitness());

    #ifdef \_DEBUG\_POPULATION

        printf("Initial CLimber\n");

        printf("Fitness: %f\n", HillClimber->getFitness());

        printf("Genes: ");

        for (int i = 0; i < NUMBER\_OF\_GENES; i++) {

            printf("%f ", HillClimber->getGenes()[i]);

        }

        printf("\n");

    #endif

    Individual \*tempClimber = new Individual();

    int zG = 0;

    while(HillClimber->getFitness() > -7){

        float \*tempGenes = HillClimber->getGenes();

        int random\_index = rand() % NUMBER\_OF\_GENES;

        tempGenes[random\_index] = (float)rand() / RAND\_MAX;

        tempClimber->setGenes(tempGenes);

        Helper::calculateFitness(tempClimber);

        if(tempClimber->getFitness() <= HillClimber->getFitness()){

            HillClimber->setGenes(tempClimber->getGenes());

            HillClimber->setFitness(tempClimber->getFitness());

        }

        Helper::fitnesses.emplace\_back(HillClimber->getFitness());

*// for (int i = 0; i < NUMBER\_OF\_GENES; i++) {*

*//     // printf("%f ", HillClimber->getGenes()[i]);*

*// }*

*// // printf("\n");*

*// printf("Fitness: %f\n", HillClimber->getFitness());*

        zG++;

    }

    printf("took %d generations\n", zG);

    printf("Fitness: %f\n", HillClimber->getFitness());

    return true;

}

### Individual.h –

#pragma once

#include <defines.h>

class Individual {

    private:

        double fitness;

        float genes[NUMBER\_OF\_GENES];

    public:

*//constructor*

        Individual();

*//destructor*

        ~Individual();

*//getters*

        float getFitness();

        float\* getGenes();

*//setters*

        void setFitness(float fitness);

        void setGenes(float genes[]);

        void setGene(int index, float value);

};

### Helper.h –

#pragma once

#include <individual.h>

#include <vector>

struct Stats {

    #ifdef FIND\_BEST

        float best;

    #else

        float worst;

    #endif

    float average;

    float total;

};

struct Generation {

    Stats population;

    Stats gladiator;

    Stats crosspoint;

    Stats mutation;

};

struct GenerationHistory {

    void\* population[POPULATION\_SIZE];

    void\* gladiators[POPULATION\_SIZE];

    void\* crosspoints[POPULATION\_SIZE];

    void\* mutations[POPULATION\_SIZE];

};

struct MetaData {

    float mutation\_rate;

    float mutation\_height;

    float tournament\_size;

    float ending\_fitness\_height;

    #ifdef PLATO\_HEIGHT

        int plato\_confirmed;

    #endif

    int generation;

    float solution\_fitness;

};

namespace Helper {

    #ifdef FIND\_BEST

        extern float best\_fitnesses[NUMBER\_OF\_GENERATIONS];

    #else

        extern float worst\_fitnesses[NUMBER\_OF\_GENERATIONS];

    #endif

    extern float average\_fitnesses[NUMBER\_OF\_GENERATIONS];

    extern Generation generations[NUMBER\_OF\_GENERATIONS];

    extern GenerationHistory generation\_history[NUMBER\_OF\_GENERATIONS];

    extern int generation\_index;

    extern int generation\_type\_index;

    extern bool cycle;

    extern int cycle\_speed;

    extern int cycle\_timer;

    extern std::vector<MetaData> meta\_data;

    extern std::vector<float> fitnesses;

    void generateGenes(Individual \*individual);

    void calculateFitness(Individual \*individual);

    void printGenes(Individual \*individual);

    void printFitness(Individual \*individual);

    void printIndividual(Individual \*individual);

    void printPopulation(void\* population[]);

    float getPopulationFitness(void\* population[]);

    float getPopulationHeight(void\* population[]);

    bool ClearPopulation(void\* population[]);

    bool CopyPopulation(void\* from[], void\* to[]);

    MetaData\* Top10Gens();

    float GetSolutionFitness(MetaData meta);

    MetaData\* GetTopSolutions();

}

### Individual.cpp –

#include <individual.h>

#include <stdio.h>

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 CONSTRUCTOR AND DESTRUCTOR*

*//*

*//      TAKES ARGUMENTS:*

*//          NONE*

*//*

*//    CREATES AN INDIVIDUAL AND PRINTS THE ADDRESS OF THE INDIVIDUAL*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

Individual::Individual() {

    #ifdef \_DEBUG\_INDIVIDUAL

        printf("0x%p constructor\n", *this*);

    #endif

}

Individual::~Individual() {

    #ifdef \_DEBUG\_INDIVIDUAL

        printf("0x%p destructor\n", *this*);

    #endif

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO GET THE FITNESS OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          NONE*

*//*

*//*

*//      RETURNS:*

*//          THE FITNESS OF THE INDIVIDUAL AS A FLOAT*

*//*

*///////////////////*

float Individual::getFitness() {

    return *this*->fitness;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO GET THE GENES OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          NONE*

*//*

*//*

*//      RETURNS:*

*//          THE GENES OF THE INDIVIDUAL AS A FLOAT ARRAY*

float\* Individual::getGenes() {

    return *this*->genes;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO SET THE FITNESS OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          fitness : THE FITNESS TO SET THE INDIVIDUAL TO*

*//*

*//*

*//      SETS THE FITNESS OF THE INDIVIDUAL*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Individual::setFitness(float fitness) {

*this*->fitness = fitness;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO SET THE GENES OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          genes : THE GENES TO SET THE INDIVIDUAL TO*

*//*

*//*

*//      cYCLE THROUGH EACH GENE IN THE INDIVIDUAL*

*//          SETS THE GENE*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Individual::setGenes(float genes[]) {

    for (int i = 0; i < NUMBER\_OF\_GENES; i++) {

*this*->genes[i] = genes[i];

    }

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO SET A GENE OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          index : THE INDEX OF THE GENE TO SET*

*//          value : THE VALUE TO SET THE GENE TO*

*//*

*//*

*//      SETS THE GENE OF THE INDIVIDUAL*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Individual::setGene(int index, float value) {

*this*->genes[index] = value;

}

### Helper.cpp –

#include <helper.h>

#include <defines.h>

#include <cmath>

#include <iostream>

#include <time.h>

namespace Helper {

        #ifdef FIND\_BEST

            float best\_fitnesses[NUMBER\_OF\_GENERATIONS];

        #else

            float worst\_fitnesses[NUMBER\_OF\_GENERATIONS];

        #endif

        float average\_fitnesses[NUMBER\_OF\_GENERATIONS];

        Generation generations[NUMBER\_OF\_GENERATIONS];

        GenerationHistory generation\_history[NUMBER\_OF\_GENERATIONS];

        int generation\_index = 0;

        int generation\_type\_index = 0;

        bool cycle = false;

        int cycle\_speed = 100;

        int cycle\_timer = 0;

        std::vector<MetaData> meta\_data;

        std::vector<float> fitnesses;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO GENERATE THE GENES OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          individual : THE INDIVIDUAL TO GENERATE THE GENES OF*

*//*

*//*

*//      CYCLES THROUGH EACH GENE IN THE INDIVIDUAL*

*//          GENERATES A RANDOM FLOAT BETWEEN 0 AND 1*

*//          SETS THE GENE TO THE RANDOM FLOAT*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Helper::generateGenes(Individual \*individual) {

        float temp\_genes[NUMBER\_OF\_GENES];

        for (int i = 0; i < NUMBER\_OF\_GENES; i++) {

            temp\_genes[i] = static\_cast<float>(std::rand()) / RAND\_MAX;

        }

    individual->setGenes(temp\_genes);

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO CALCULATE THE FITNESS OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          individual : THE INDIVIDUAL TO CALCULATE THE FITNESS OF*

*//*

*//*

*//      CYCLES THROUGH EACH GENE IN THE INDIVIDUAL*

*//          ADDS THE GENE TO THE FITNESS*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Helper::calculateFitness(Individual \*individual) {

*//calculate fitness aka test function*

    float fitness = 0;

    float \*genes = individual->getGenes();

    #ifdef SIMPLE\_TEST

        for (int i = 0; i < NUMBER\_OF\_GENES; i++) {

            fitness += genes[i];

        }

    #endif

    #ifdef ADVANCED\_TEST

        float temp = 0;

        for (int i = 1; i < NUMBER\_OF\_GENES; i++) {

            fitness += i \* (pow(((2 \* pow(genes[i], 2)) - genes[i-1]), 2));

        }

        fitness += pow((genes[0]-1), 2);

    #endif

    #ifdef COMPLEX\_TEST

        float temp = 0, temp2 = 0;

        for (int i = 0; i < NUMBER\_OF\_GENES; i++) {

            temp += pow(genes[i], 2);

            temp2 += 0.5 \* (i+1) \* genes[i];

        }

        fitness = temp + pow(temp2, 2) + pow(temp2, 4);

    #endif

    #ifdef TRID\_FUNCTION

        float temp = 0, temp2 = 0;

        for (int i = 0; i < NUMBER\_OF\_GENES; i++) {

            temp += pow(genes[i] - 1, 2);

        }

        for (int i = 1; i < NUMBER\_OF\_GENES; i++) {

            temp2 += genes[i] \* genes[i-1];

        }

        fitness = temp - temp2;

*// for (int i = 0; i < NUMBER\_OF\_GENES; i++) {*

*//     fitness += i \* pow(genes[i], 2);*

*// }*

    #endif

    individual->setFitness(fitness);

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO PRINT THE GENES OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          individual : THE INDIVIDUAL TO PRINT THE GENES OF*

*//*

*//*

*//      CYCLES THROUGH EACH GENE IN THE INDIVIDUAL*

*//          PRINTS THE GENE*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Helper::printGenes(Individual \*individual) {

    float \*genes = individual->getGenes();

    std::cout << "Genes: ";

    for (int i = 0; i < NUMBER\_OF\_GENES; i++) {

        std::cout << genes[i] << " ";

    }

    std::cout << ";" << std::endl;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO PRINT THE FITNESS OF AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          individual : THE INDIVIDUAL TO PRINT THE FITNESS OF*

*//*

*//*

*//      PRINTS THE FITNESS OF THE INDIVIDUAL*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Helper::printFitness(Individual \*individual) {

*//print fitness*

    float fitness = individual->getFitness();

    std::cout << "Fitness: " << fitness << std::endl;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO PRINT AN INDIVIDUAL*

*//      TAKES ARGUMENTS:*

*//          individual : THE INDIVIDUAL TO PRINT*

*//*

*//*

*//      PRINTS THE GENES AND FITNESS OF THE INDIVIDUAL*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Helper::printIndividual(Individual \*individual) {

    std::cout << "\n\nIndividual: " << std::endl;

*//print genes and fitness*

    printGenes(individual);

    printFitness(individual);

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO PRINT A POPULATION*

*//      TAKES ARGUMENTS:*

*//          population : THE POPULATION TO PRINT*

*//*

*//*

*//      CYCLES THROUGH EACH INDIVIDUAL IN THE POPULATION*

*//          PRINTS THE INDIVIDUAL*

*//*

*//      RETURNS:*

*//          NONE*

*//*

*///////////////////*

void Helper::printPopulation(void\* population[]) {

    std::cout << "Population: " << std::endl;

*//print population*

    for (int i = 0; i < POPULATION\_SIZE; i++) {

        Individual \*individual = (Individual\*)population[i];

        printIndividual(individual);

    }

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO GET THE FITNESS OF A POPULATION*

*//      TAKES ARGUMENTS:*

*//          population : THE POPULATION TO GET THE FITNESS OF*

*//*

*//*

*//      CYCLES THROUGH EACH INDIVIDUAL IN THE POPULATION*

*//          GETS THE FITNESS OF EACH INDIVIDUAL*

*//          ADDS THE FITNESS OF EACH INDIVIDUAL TO THE FITNESS OF THE POPULATION*

*//*

*//      RETURNS:*

*//          fitness : THE FITNESS OF THE POPULATION*

*//*

*///////////////////*

float Helper::getPopulationFitness(void\* population[]) {

*//calculate fitness of population*

    float fitness = 0;

    for (int i = 0; i < POPULATION\_SIZE; i++) {

        Individual \*individual = (Individual\*)population[i];

        fitness += individual->getFitness();

    }

    return fitness;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 FUNCTION TO GET THE HEIGHT OF A POPULATION*

*//      TAKES ARGUMENTS:*

*//          population : THE POPULATION TO GET THE HEIGHT OF*

*//*

*//*

*//      CYCLES THROUGH EACH INDIVIDUAL IN THE POPULATION*

*//          GETS THE FITNESS OF EACH INDIVIDUAL*

*//*

*//          IF FINDING BEST*

*//              IF THE FITNESS IS GREATER THAN THE CURRENT HEIGHT*

*//                  SET THE HEIGHT TO THE FITNESS*

*//          IF FINDING WORST*

*//              IF THE FITNESS IS LESS THAN THE CURRENT HEIGHT*

*//                  SET THE HEIGHT TO THE FITNESS*

*//*

*//      RETURNS:*

*//          height : THE HEIGHT OF THE POPULATION*

*//*

*///////////////////*

float Helper::getPopulationHeight(void\* population[]) {

*//calculate fitness of population*

    float height;

    Individual \*individual = (Individual\*)population[0];

    height = individual->getFitness();

    #ifdef FIND\_BEST

        for (int i = 1; i < POPULATION\_SIZE; i++) {

            Individual \*individual = (Individual\*)population[i];

            if (individual->getFitness() > height) {

                height = individual->getFitness();

            }

        }

    #else

            for (int i = 1; i < POPULATION\_SIZE; i++) {

                Individual \*individual = (Individual\*)population[i];

                if (individual->getFitness() < height) {

                    height = individual->getFitness();

                }

            }

    #endif

    return height;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 SIMPLE FUNCTION TO CLEAR A POPULATION*

*//      TAKES ARGUMENTS:*

*//          population : THE POPULATION TO CLEAR*

*//*

*//*

*//      CYCLES THROUGH EACH INDIVIDUAL IN THE POPULATION AND DELETES IT*

*//          THEN SETS THE POINTER TO NULL*

*//*

*//      RETURNS:*

*//          true : IF THE POPULATION WAS CLEARED*

*//          false : IF THE POPULATION WAS NOT CLEARED*

*///////////////////*

bool Helper::ClearPopulation(void\* population[]){

    for (int i = 0; i < POPULATION\_SIZE; i++) {

        delete (Individual\*)population[i];

        population[i] = NULL;

    }

    return true;

}

*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*//*

*//                                 SIMPLE FUNCTION TO COPY A POPULATION TO ANOTHER POPULATION*

*//      TAKES ARGUMENTS:*

*//          from : THE POPULATION TO COPY FROM*

*//          to : THE POPULATION TO COPY TO*

*//*

*//*

*//      COPIES THE POPULATION FROM ONE POPULATION TO ANOTHER BY CYCLING THROUGH EACH INDIVIDUAL*

*//          AND COPYING THE GENES AND FITNESS OF EACH INDIVIDUAL TO A NEW INSTANCE OF AN INDIVIDUAL*

*//          AND ADDING A POINTER TO IT TO THE NEW POPULATION ARRAY*

*//*

*//      RETURNS:*

*//          true : IF THE POPULATION WAS COPIED*

*//          false : IF THE POPULATION WAS NOT COPIED*

*//*

*///////////////////*

bool Helper::CopyPopulation(void\* from[], void\* to[]){

    for (int i = 0; i < POPULATION\_SIZE; i++) {

        Individual \*individual = new Individual();

        individual->setGenes(((Individual\*)from[i])->getGenes());

        individual->setFitness(((Individual\*)from[i])->getFitness());

        to[i] = individual;

    }

    return true;

}

MetaData\* Helper::Top10Gens(){

    MetaData\* top10 = new MetaData[NUMBER\_OF\_RESULTS];

    for (int i = 0; i < NUMBER\_OF\_RESULTS; i++) {

        top10[i] = meta\_data[i];

    }

*// find the best 10 candidates from meta\_data and add to top10*

    for (int i = NUMBER\_OF\_RESULTS - 1; i < meta\_data.size(); i++) {

        if (meta\_data[i].ending\_fitness\_height == 0) {

            continue;

        }

        for (int j = 0; j < NUMBER\_OF\_RESULTS; j++) {

            #ifdef FIND\_BEST

                if (meta\_data[i].ending\_fitness\_height > top10[j].ending\_fitness\_height) {

                    top10[j] = meta\_data[i];

                    break;

                }

            #else

                if (meta\_data[i].ending\_fitness\_height < top10[j].ending\_fitness\_height) {

                    top10[j] = meta\_data[i];

                    break;

                }

            #endif

        }

    }

    return top10;

}

float Helper::GetSolutionFitness(MetaData meta){

    float solution\_fitness = 0;

    int gen\_loops, glad\_loops, cross\_loops, mut\_loops;

*// if(meta.plato\_confirmed != 0){*

*//  gen\_loops = meta.plato\_confirmed \* POPULATION\_SIZE;*

*// }*

*// else{*

*//  gen\_loops = NUMBER\_OF\_GENERATIONS \* POPULATION\_SIZE;*

*// }*

*// glad\_loops = gen\_loops \* (TOURNAMENT\_SIZE \* TOURNAMENT\_SIZE - 1);*

*// cross\_loops = (gen\_loops \* (POPULATION\_SIZE / 2)) \* NUMBER\_OF\_GENES;*

*// mut\_loops = (gen\_loops \* POPULATION\_SIZE) \* NUMBER\_OF\_GENES;*

*// solution\_fitness = (gen\_loops + glad\_loops + cross\_loops + mut\_loops) \* meta.ending\_fitness\_height;*

    if(meta.plato\_confirmed != 0){

        gen\_loops = meta.plato\_confirmed;

    }   else    {

        gen\_loops = NUMBER\_OF\_GENERATIONS;

    }

    glad\_loops = (TOURNAMENT\_SIZE \* TOURNAMENT\_SIZE) \* POPULATION\_SIZE;

    cross\_loops = (POPULATION\_SIZE / 2) \* NUMBER\_OF\_GENES;

    mut\_loops = POPULATION\_SIZE \* NUMBER\_OF\_GENES;

    solution\_fitness = (gen\_loops \* (glad\_loops + cross\_loops + mut\_loops)) \* meta.ending\_fitness\_height;*//+ 0.0000001  sqrt(pow(meta.ending\_fitness\_height, 2))*

    return solution\_fitness;

}

MetaData\* Helper::GetTopSolutions(){

    MetaData\* top10sols = new MetaData[NUMBER\_OF\_RESULTS];

    for (int i = 0; i < NUMBER\_OF\_RESULTS; i++) {

        top10sols[i] = meta\_data[i];

    }

*// find the best 10 candidates from meta\_data and add to top10*

    for (int i = NUMBER\_OF\_RESULTS - 1; i < meta\_data.size(); i++) {

        if(meta\_data[i].solution\_fitness == 0){

            continue;

        }

        for (int j = 0; j < NUMBER\_OF\_RESULTS; j++) {

            #ifdef FIND\_BEST

                if (meta\_data[i].solution\_fitness > top10sols[j].solution\_fitness) {

                    top10sols[j] = meta\_data[i];

                    break;

                }

            #else

                if (meta\_data[i].solution\_fitness < top10sols[j].solution\_fitness) {

                    top10sols[j] = meta\_data[i];

                    break;

                }

            #endif

        }

    }

    return top10sols;

}

### Graphing.cpp –

#include <graphing.h>

#include <helper.h>

#include <defines.h>

#include <imgui.h>

#include <imgui\_impl\_dx9.h>

#include <imgui\_impl\_win32.h>

#include <imguipp.h>

#include <implot.h>

#include <implot\_internal.h>

#include <string.h>

#include <thread>

int menu\_index = 0;

int lookup\_index = 0;

extern IMGUI\_IMPL\_API LRESULT ImGui\_ImplWin32\_WndProcHandler(

    HWND window,

    UINT message,

    WPARAM wParam,

    LPARAM lParam

);

LRESULT \_\_stdcall WindowProcess(HWND window, UINT message, WPARAM wParam, LPARAM lParam)

{

    if (ImGui\_ImplWin32\_WndProcHandler(window, message, wParam, lParam))

        return TRUE;

    switch (message)

    {

        case WM\_SIZE:

        {

            if (gui::device && wParam != SIZE\_MINIMIZED)

            {

                gui::presentParameters.BackBufferWidth = LOWORD(lParam);

                gui::presentParameters.BackBufferHeight = HIWORD(lParam);

                gui::ResetDevice();

            }

            return 0;

        }

        case WM\_SYSCOMMAND:

        {

            if ((wParam & 0xfff0) == SC\_KEYMENU)

                return 0;

        }

        break;

        case WM\_DESTROY:

        {

            PostQuitMessage(0);

        }

        return 0;

        case WM\_LBUTTONDOWN:

        {

            gui::position = MAKEPOINTS(lParam);

        }

        return 0;

        case WM\_MOUSEMOVE:

        {

            if (wParam == MK\_LBUTTON)

            {

                POINTS points = MAKEPOINTS(lParam);

                RECT rect = {};

                GetWindowRect(gui::window, &rect);

                rect.left += points.x - gui::position.x;

                rect.top += points.y - gui::position.y;

                if (gui::position.x >= 0 &&

                    gui::position.x <= gui::WIDTH &&

                    gui::position.y >= 0 && gui::position.y <= 19)

                {

                    SetWindowPos(

                        gui::window,

                        HWND\_TOPMOST,

                        rect.left,

                        rect.top,

                        0,

                        0,

                        SWP\_SHOWWINDOW | SWP\_NOSIZE | SWP\_NOZORDER

                    );

                }

            }

        }

        return 0;

    }

    return DefWindowProcW(window, message, wParam, lParam);

}

void gui::CreateHWindow(

    const char\* windowName,

    const char\* className) noexcept

{

    windowClass.cbSize = sizeof(WNDCLASSEXA);

    windowClass.style = CS\_CLASSDC;

    windowClass.lpfnWndProc = WindowProcess;

    windowClass.cbClsExtra = 0;

    windowClass.cbWndExtra = 0;

    windowClass.hInstance = GetModuleHandleA(0);

    windowClass.hIcon = 0;

    windowClass.hCursor = 0;

    windowClass.hbrBackground = 0;

    windowClass.lpszMenuName = 0;

    windowClass.lpszClassName = className;

    windowClass.hIconSm = 0;

    RegisterClassExA(&windowClass);

    window = CreateWindowA(

        className,

        windowName,

        WS\_POPUP,

        100,

        100,

        WIDTH,

        HEIGHT,

        0,

        0,

        windowClass.hInstance,

        0

    );

    ShowWindow(window, SW\_SHOWDEFAULT);

    UpdateWindow(window);

}

void gui::DestroyHWindow() noexcept

{

    DestroyWindow(window);

    UnregisterClassA(windowClass.lpszClassName, windowClass.hInstance);

}

bool gui::CreateDevice() noexcept

{

    d3d = Direct3DCreate9(D3D\_SDK\_VERSION);

    if (!d3d)

        return false;

    ZeroMemory(&presentParameters, sizeof(presentParameters));

    presentParameters.Windowed = true;

    presentParameters.SwapEffect = D3DSWAPEFFECT\_DISCARD;

    presentParameters.BackBufferFormat = D3DFMT\_UNKNOWN;

    presentParameters.EnableAutoDepthStencil = true;

    presentParameters.AutoDepthStencilFormat = D3DFMT\_D16;

    presentParameters.PresentationInterval = D3DPRESENT\_INTERVAL\_ONE;

    if (d3d->CreateDevice(

        D3DADAPTER\_DEFAULT,

        D3DDEVTYPE\_HAL,

        window,

        D3DCREATE\_HARDWARE\_VERTEXPROCESSING,

        &presentParameters,

        &device) <0 )

        return false;

    return true;

}

void gui::ResetDevice() noexcept

{

    ImGui\_ImplDX9\_InvalidateDeviceObjects();

    const auto result = device->Reset(&presentParameters);

    if (result == D3DERR\_INVALIDCALL)

        IM\_ASSERT(0);

    ImGui\_ImplDX9\_CreateDeviceObjects();

}

void gui::DestroyDevice() noexcept

{

    if (device)

    {

        device->Release();

        device = nullptr;

    }

    if (d3d)

    {

        d3d->Release();

        d3d = nullptr;

    }

}

void gui::CreateImGui() noexcept

{

    IMGUI\_CHECKVERSION();

    ImGui::CreateContext();

    ImPlot::CreateContext();

    ImGuiIO& io = ::ImGui::GetIO();

    io.IniFilename = NULL;

    ImGui::StyleColorsDark();

    ImGui\_ImplWin32\_Init(window);

    ImGui\_ImplDX9\_Init(device);

}

void gui::DestroyImGui() noexcept

{

    ImGui\_ImplDX9\_Shutdown();

    ImGui\_ImplWin32\_Shutdown();

    ImPlot::DestroyContext();

    ImGui::DestroyContext();

}

void gui::BeginRender() noexcept

{

    MSG message;

    while (PeekMessage(&message, 0, 0, 0, PM\_REMOVE))

    {

        TranslateMessage(&message);

        DispatchMessage(&message);

    }

    ImGui\_ImplDX9\_NewFrame();

    ImGui\_ImplWin32\_NewFrame();

    ImGui::NewFrame();

}

void gui::EndRender() noexcept

{

    ImGui::EndFrame();

    device->SetRenderState(D3DRS\_ZENABLE, FALSE);

    device->SetRenderState(D3DRS\_ALPHABLENDENABLE, FALSE);

    device->SetRenderState(D3DRS\_SCISSORTESTENABLE, FALSE);

    device->Clear(0, 0, D3DCLEAR\_TARGET | D3DCLEAR\_ZBUFFER, D3DCOLOR\_RGBA(0, 0, 0, 255), 1.0f, 0);

    if (device->BeginScene() >= 0)

    {

        ImGui::Render();

        ImGui\_ImplDX9\_RenderDrawData(ImGui::GetDrawData());

        device->EndScene();

    }

    const auto result = device->Present(0, 0, 0, 0);

    if (result == D3DERR\_DEVICELOST && device->TestCooperativeLevel() == D3DERR\_DEVICENOTRESET)

        ResetDevice();

}

void gui::Theme() noexcept

{

    ImGuiStyle\* style = &ImGui::GetStyle();

    style->WindowTitleAlign = ImVec2(0.5f, 0.5f);

    style->WindowMinSize = ImVec2(WIDTH, HEIGHT);

    style->FramePadding = ImVec2(8, 6);

    style->Colors[ImGuiCol\_TitleBg] = { 0.32f, 0.00f, 0.47f, 1.00f };

    style->Colors[ImGuiCol\_TitleBgActive] = { 0.32f, 0.00f, 0.47f, 1.00f };

    style->Colors[ImGuiCol\_TitleBgCollapsed] = { 0.00f, 0.00f, 0.00f, 1.00f };

    style->Colors[ImGuiCol\_Button] = { 0.30f, 0.00f, 0.13f, 1.00f };

    style->Colors[ImGuiCol\_ButtonHovered] = { 0.12f, 0.11f, 0.12f, 1.00f };

    style->Colors[ImGuiCol\_ButtonActive] = { 0.71f, 0.00f, 0.29f, 1.00f };

    style->Colors[ImGuiCol\_Separator] = { 0.30f, 0.00f, 0.13f, 1.00f };

    style->Colors[ImGuiCol\_SeparatorHovered] = { 0.12f, 0.11f, 0.12f, 1.00f };

    style->Colors[ImGuiCol\_SeparatorActive] = { 0.71f, 0.00f, 0.29f, 1.00f };

    style->Colors[ImGuiCol\_Text] = { 1.00f, 1.00f, 1.00f, 1.00f };

    style->Colors[ImGuiCol\_ChildBg] = { 0.00f, 0.00f, 0.00f, 0.00f };

    style->Colors[ImGuiCol\_WindowBg] = { 0.06f, 0.06f, 0.06f, 0.94f };

    style->Colors[ImGuiCol\_Border] = { 0.27f, 0.00f, 0.11f, 1.00f };

    style->Colors[ImGuiCol\_FrameBg] = { 0.30f, 0.00f, 0.13f, 1.00f };

    style->Colors[ImGuiCol\_FrameBgHovered] = { 0.12f, 0.11f, 0.12f, 1.00f };

    style->Colors[ImGuiCol\_FrameBgActive] = { 0.30f, 0.00f, 0.13f, 1.00f };

    style->Colors[ImGuiCol\_CheckMark] = { 0.71f, 0.00f, 0.29f, 1.00f };

    style->Colors[ImGuiCol\_SliderGrab] = { 0.30f, 0.00f, 0.13f, 1.00f };

    style->Colors[ImGuiCol\_SliderGrabActive] = { 0.71f, 0.00f, 0.29f, 1.00f };

    style->Colors[ImGuiCol\_ScrollbarBg] = { 0.30f, 0.00f, 0.13f, 1.00f };

    style->Colors[ImGuiCol\_ScrollbarGrab] = { 0.71f, 0.00f, 0.29f, 1.00f };

    style->Colors[ImGuiCol\_ScrollbarGrabHovered] = { 0.12f, 0.11f, 0.12f, 1.00f };

    style->Colors[ImGuiCol\_ScrollbarGrabActive] = { 0.71f, 0.00f, 0.29f, 1.00f };

    style->Colors[ImGuiCol\_Header] = { 0.30f, 0.00f, 0.13f, 1.00f };

    style->Colors[ImGuiCol\_HeaderHovered] = { 0.12f, 0.11f, 0.12f, 1.00f };

    style->Colors[ImGuiCol\_HeaderActive] = { 0.30f, 0.00f, 0.13f, 1.00f };

}

void gui::Menu() noexcept

{

#ifdef RUN\_HILL\_CLIMBER

    ImPlot::BeginPlot("Hill Climber");

    {

        float fitnesses[NUMBER\_OF\_GENERATIONS];

        for (int i = 0; i < NUMBER\_OF\_GENERATIONS; i++) {

            fitnesses[i] = Helper::fitnesses[i];

        }

        ImPlot::PlotLine("Fitness", fitnesses, NUMBER\_OF\_GENERATIONS);

    }

    ImPlot::EndPlot();

#endif

#ifdef RUN\_GENETIC\_ALGORITHM

    ImPlot::BeginPlot("Winning Generations");

    {

        ImPlot::PlotLine("Average", Helper::average\_fitnesses, NUMBER\_OF\_GENERATIONS);

        #ifdef FIND\_BEST

            ImPlot::PlotLine("Best", Helper::best\_fitnesses, NUMBER\_OF\_GENERATIONS);

        #else

            ImPlot::PlotLine("Worst", Helper::worst\_fitnesses, NUMBER\_OF\_GENERATIONS);

        #endif

    }

    ImPlot::EndPlot();

    #ifdef \_PLOT\_GRAPHS

            ImPlot::BeginPlot("Population");

            {

                #ifdef \_GRAPH\_POPULATION

                    #ifdef FIND\_BEST

                        float population\_best[NUMBER\_OF\_GENERATIONS];

                    #else

                        float population\_worst[NUMBER\_OF\_GENERATIONS];

                    #endif

                    float population\_average[NUMBER\_OF\_GENERATIONS];

                    for (int i = 0; i < NUMBER\_OF\_GENERATIONS; i++) {

                        #ifdef FIND\_BEST

                            population\_best[i] = Helper::generations[i].population.best;

                        #else

                            population\_worst[i] = Helper::generations[i].population.worst;

                        #endif

                        population\_average[i] = Helper::generations[i].population.average;

                    }

                    #ifdef FIND\_BEST

                        ImPlot::PlotLine("Population Best", population\_best, NUMBER\_OF\_GENERATIONS);

                    #else

                        ImPlot::PlotLine("Population Worst", population\_worst, NUMBER\_OF\_GENERATIONS);

                    #endif

                    ImPlot::PlotLine("Population Average", population\_average, NUMBER\_OF\_GENERATIONS);

                #endif

                #ifdef \_GRAPH\_GLADIATOR

                    #ifdef FIND\_BEST

                        float gladiator\_best[NUMBER\_OF\_GENERATIONS];

                    #else

                        float gladiator\_worst[NUMBER\_OF\_GENERATIONS];

                    #endif

                    float gladiator\_average[NUMBER\_OF\_GENERATIONS];

                    for (int i = 0; i < NUMBER\_OF\_GENERATIONS; i++) {

                        #ifdef FIND\_BEST

                            gladiator\_best[i] = Helper::generations[i].gladiator.best;

                        #else

                            gladiator\_worst[i] = Helper::generations[i].gladiator.worst;

                        #endif

                        gladiator\_average[i] = Helper::generations[i].gladiator.average;

                    }

                    #ifdef FIND\_BEST

                        ImPlot::PlotLine("Gladiator Best", gladiator\_best, NUMBER\_OF\_GENERATIONS);

                    #else

                        ImPlot::PlotLine("Gladiator Worst", gladiator\_worst, NUMBER\_OF\_GENERATIONS);

                    #endif

                    ImPlot::PlotLine("Gladiator Average", gladiator\_average, NUMBER\_OF\_GENERATIONS);

                #endif

                #ifdef \_GRAPH\_CROSSPOINT

                    #ifdef FIND\_BEST

                        float crosspoint\_best[NUMBER\_OF\_GENERATIONS];

                    #else

                        float crosspoint\_worst[NUMBER\_OF\_GENERATIONS];

                    #endif

                    float crosspoint\_average[NUMBER\_OF\_GENERATIONS];

                    for (int i = 0; i < NUMBER\_OF\_GENERATIONS; i++) {

                        #ifdef FIND\_BEST

                            crosspoint\_best[i] = Helper::generations[i].crosspoint.best;

                        #else

                            crosspoint\_worst[i] = Helper::generations[i].crosspoint.worst;

                        #endif

                        crosspoint\_average[i] = Helper::generations[i].crosspoint.average;

                    }

                    #ifdef FIND\_BEST

                        ImPlot::PlotLine("Crosspoint Best", crosspoint\_best, NUMBER\_OF\_GENERATIONS);

                    #else

                        ImPlot::PlotLine("Crosspoint Worst", crosspoint\_worst, NUMBER\_OF\_GENERATIONS);

                    #endif

                    ImPlot::PlotLine("Crosspoint Average", crosspoint\_average, NUMBER\_OF\_GENERATIONS);

                #endif

                #ifdef \_GRAPH\_MUTATION

                    #ifdef FIND\_BEST

                        float mutation\_best[NUMBER\_OF\_GENERATIONS];

                    #else

                        float mutation\_worst[NUMBER\_OF\_GENERATIONS];

                    #endif

                    float mutation\_average[NUMBER\_OF\_GENERATIONS];

                    for (int i = 0; i < NUMBER\_OF\_GENERATIONS; i++) {

                        #ifdef FIND\_BEST

                            mutation\_best[i] = Helper::generations[i].mutation.best;

                        #else

                            mutation\_worst[i] = Helper::generations[i].mutation.worst;

                        #endif

                        mutation\_average[i] = Helper::generations[i].mutation.average;

                    }

                    #ifdef FIND\_BEST

                        ImPlot::PlotLine("Mutation Best", mutation\_best, NUMBER\_OF\_GENERATIONS);

                    #else

                        ImPlot::PlotLine("Mutation Worst", mutation\_worst, NUMBER\_OF\_GENERATIONS);

                    #endif

                    ImPlot::PlotLine("Mutation Average", mutation\_average, NUMBER\_OF\_GENERATIONS);

                #endif

            }

            ImPlot::EndPlot();

        #ifdef \_GRAPH\_GENERATION\_HISTORY

            ImGui::SeparatorEx(ImGuiSeparatorFlags\_Horizontal);

            ImGui::SliderInt("Population Type", &Helper::generation\_type\_index, 0, 3);

            ImGui::SameLine();

            ImGui::Button("Previous", ImVec2(100, 20));

            {

                if (ImGui::IsItemClicked()) {

                    if(Helper::generation\_type\_index > 0){

                        Helper::generation\_type\_index--;

                    } else {

                        Helper::generation\_type\_index = 3;

                        Helper::generation\_index--;

                        if(Helper::generation\_index < 0)

                            Helper::generation\_index = (NUMBER\_OF\_GENERATIONS - 1);

                    }

                }

            }

            ImGui::SameLine();

            ImGui::Button("Next", ImVec2(100,20));

            {

                if (ImGui::IsItemClicked()) {

                    if (Helper::generation\_type\_index < 3) {

                        Helper::generation\_type\_index++;

                    }

                    else {

                        Helper::generation\_type\_index = 0;

                        Helper::generation\_index++;

                        if (Helper::generation\_index > (NUMBER\_OF\_GENERATIONS - 1))

                            Helper::generation\_index = 0;

                    }

                }

            }

            ImGui::SliderInt("Generation", &Helper::generation\_index, 0, NUMBER\_OF\_GENERATIONS - 1);

            ImGui::Button("Start Cycle", ImVec2(100, 20));

            {

                if (ImGui::IsItemClicked()) {

                    Helper::cycle = true;

                }

            }

            ImGui::SameLine();

            ImGui::Button("Stop Cycle", ImVec2(100, 20));

            {

                if (ImGui::IsItemClicked()) {

                    Helper::cycle = false;

                }

            }

            ImGui::SliderInt("Cycle Speed", &Helper::cycle\_speed, 120, 1);

            if(Helper::cycle){

                Helper::cycle\_timer++;

                if (Helper::cycle\_timer > Helper::cycle\_speed) {

                    Helper::cycle\_timer = 0;

                    Helper::generation\_type\_index++;

                    if (Helper::generation\_type\_index > 3) {

                        Helper::generation\_type\_index = 0;

                        Helper::generation\_index++;

                        if (Helper::generation\_index > (NUMBER\_OF\_GENERATIONS - 1))

                            Helper::generation\_index = 0;

                    }

                }

            }

            float generation\_fitness\_height[POPULATION\_SIZE];

            float generation\_fitness\_average[POPULATION\_SIZE];

            float individual\_fitness\_average[POPULATION\_SIZE];

*// ImPlot::SetNextAxesToFit();*

            switch (Helper::generation\_type\_index){

                case 0:

                    ImPlot::BeginPlot("Populations");

                    {

                        for (int k = 0; k < POPULATION\_SIZE; k++) {

                            Individual\* individual = static\_cast<Individual\*>(Helper::generation\_history[Helper::generation\_index].population[k]);

                            generation\_fitness\_height[k] = individual->getFitness();

                            generation\_fitness\_average[k] = Helper::generations[Helper::generation\_index].population.average;

                            individual\_fitness\_average[k] = (individual->getFitness() / NUMBER\_OF\_GENES);

                        }

                        ImPlot::PlotLine("Fitness", generation\_fitness\_height, POPULATION\_SIZE);

                        ImPlot::PlotLine("Height Average", generation\_fitness\_average, POPULATION\_SIZE);

                        ImPlot::PlotLine("Individual Average", individual\_fitness\_average, POPULATION\_SIZE);

                    }

                    ImPlot::EndPlot();

                    break;

                case 1:

                    ImPlot::BeginPlot("Gladiators");

                    {

                        for (int k = 0; k < POPULATION\_SIZE; k++) {

                            Individual\* individual = static\_cast<Individual\*>(Helper::generation\_history[Helper::generation\_index].gladiators[k]);

                            generation\_fitness\_height[k] = individual->getFitness();

                            generation\_fitness\_average[k] = Helper::generations[Helper::generation\_index].gladiator.average;

                            individual\_fitness\_average[k] = individual->getFitness() / NUMBER\_OF\_GENES;

                        }

                        ImPlot::PlotLine("Fitness", generation\_fitness\_height, POPULATION\_SIZE);

                        ImPlot::PlotLine("Average", generation\_fitness\_average, POPULATION\_SIZE);

                        ImPlot::PlotLine("Individual Average", individual\_fitness\_average, POPULATION\_SIZE);

                    }

                    ImPlot::EndPlot();

                    break;

                case 2:

                    ImPlot::BeginPlot("Crosspoints");

                    {

                        for (int k = 0; k < POPULATION\_SIZE; k++) {

                            Individual\* individual = static\_cast<Individual\*>(Helper::generation\_history[Helper::generation\_index].crosspoints[k]);

                            generation\_fitness\_height[k] = individual->getFitness();

                            generation\_fitness\_average[k] = Helper::generations[Helper::generation\_index].crosspoint.average;

                            individual\_fitness\_average[k] = individual->getFitness() / NUMBER\_OF\_GENES;

                        }

                        ImPlot::PlotLine("Fitness", generation\_fitness\_height, POPULATION\_SIZE);

                        ImPlot::PlotLine("Average", generation\_fitness\_average, POPULATION\_SIZE);

                        ImPlot::PlotLine("Individual Average", individual\_fitness\_average, POPULATION\_SIZE);

                    }

                    ImPlot::EndPlot();

                    break;

                case 3:

                    ImPlot::BeginPlot("Mutations");

                    {

                        for (int k = 0; k < POPULATION\_SIZE; k++) {

                            Individual\* individual = static\_cast<Individual\*>(Helper::generation\_history[Helper::generation\_index].mutations[k]);

                            generation\_fitness\_height[k] = individual->getFitness();

                            generation\_fitness\_average[k] = Helper::generations[Helper::generation\_index].mutation.average;

                            individual\_fitness\_average[k] = individual->getFitness() / NUMBER\_OF\_GENES;

                        }

                        ImPlot::PlotLine("Fitness", generation\_fitness\_height, POPULATION\_SIZE);

                        ImPlot::PlotLine("Average", generation\_fitness\_average, POPULATION\_SIZE);

                        ImPlot::PlotLine("Individual Average", individual\_fitness\_average, POPULATION\_SIZE);

                    }

                    ImPlot::EndPlot();

                    break;

            }

        #endif

    #endif

    #ifdef \_META\_AI

        ImGui::SeparatorEx(ImGuiSeparatorFlags\_Horizontal);

        ImGui::Button("View Generalised Data", ImVec2(ImGui::GetContentRegionAvail().x/3 - 10, 20));

        {

            if (ImGui::IsItemClicked()) {

                menu\_index = 0;

            }

        }

        ImGui::SameLine();

        ImGui::Button("View Arguments By Fitness", ImVec2(ImGui::GetContentRegionAvail().x / 3 - 10, 20));

        {

            if (ImGui::IsItemClicked()) {

                menu\_index = 1;

            }

        }

        ImGui::SameLine();

        ImGui::Button("View Plato preformance", ImVec2(ImGui::GetContentRegionAvail().x / 3 - 10, 20));

        {

            if (ImGui::IsItemClicked()) {

                menu\_index = 2;

            }

        }

        ImGui::SameLine();

        ImGui::Button("View Combined Plato preformance", ImVec2(ImGui::GetContentRegionAvail().x / 3 - 10, 20));

        {

            if (ImGui::IsItemClicked()) {

                menu\_index = 3;

            }

        }

        ImGui::SameLine();

        ImGui::Button("View Tabled Results", ImVec2(ImGui::GetContentRegionAvail().x / 3 - 10, 20));

        {

            if (ImGui::IsItemClicked()) {

                menu\_index = 4;

            }

        }

        ImGui::SameLine();

        ImGui::Button("View Solution Fitness", ImVec2(ImGui::GetContentRegionAvail().x / 3 - 10, 20));

        {

            if (ImGui::IsItemClicked()) {

                menu\_index = 5;

            }

        }

        switch (menu\_index)

        {

        case 0: {

                ImPlot::BeginPlot("Meta AI");

                {

                    for (int i = 0; i < Helper::meta\_data.size(); i++) {

                        ImPlot::SetNextMarkerStyle(ImPlotMarker\_Circle);

                        ImPlot::PlotStems("Mutation Rate", &Helper::meta\_data[i].ending\_fitness\_height, &Helper::meta\_data[i].mutation\_rate, 1);

                        ImPlot::SetNextMarkerStyle(ImPlotMarker\_Square);

                        ImPlot::PlotStems("Mutation Height", &Helper::meta\_data[i].ending\_fitness\_height, &Helper::meta\_data[i].mutation\_height, 1);

                        ImPlot::SetNextMarkerStyle(ImPlotMarker\_Diamond);

                        ImPlot::PlotStems("Tournament Size", &Helper::meta\_data[i].ending\_fitness\_height, &Helper::meta\_data[i].tournament\_size, 1);

                    }

                }

                ImPlot::EndPlot();

            break;

        }

        case 1: {

            ImPlot::BeginPlot("View Args by Fitness");

            {

                lookup\_index = ImPlot::GetPlotMousePos().x;

                if (lookup\_index < 0)

                    lookup\_index = 0;

                if (lookup\_index > Helper::meta\_data.size() - 1)

                    lookup\_index = Helper::meta\_data.size() - 1;

                std::vector<float> ending\_fitnesses;

                for (int i = 0; i < Helper::meta\_data.size(); i++){

                    ending\_fitnesses.emplace\_back(Helper::meta\_data[i].ending\_fitness\_height);

                }

                ImPlot::SetNextMarkerStyle(ImPlotMarker\_Circle);

                ImPlot::PlotStems("Ending Fitnesses", ending\_fitnesses.data(), Helper::meta\_data.size());

            }

            ImPlot::EndPlot();

            break;

        }

        case 2: {

            ImPlot::BeginPlot("Plato Preformance");

            {

                lookup\_index = ImPlot::GetPlotMousePos().x;

                if (lookup\_index < 0)

                    lookup\_index = 0;

                if (lookup\_index > Helper::meta\_data.size() - 1)

                    lookup\_index = Helper::meta\_data.size() - 1;

                std::vector<float> plato\_confirmed;

                for (int i = 0; i < Helper::meta\_data.size(); i++) {

                    plato\_confirmed.emplace\_back(Helper::meta\_data[i].plato\_confirmed);

                }

                ImPlot::SetNextMarkerStyle(ImPlotMarker\_Circle);

                ImPlot::PlotStems("Plato Confirmed", plato\_confirmed.data(), Helper::meta\_data.size());

            }

            ImPlot::EndPlot();

            break;

        }

        case 3: {

            ImPlot::BeginPlot("Combined Plato Preformance");

            {

                lookup\_index = ImPlot::GetPlotMousePos().x;

                if (lookup\_index < 0)

                    lookup\_index = 0;

                if (lookup\_index > Helper::meta\_data.size() - 1)

                    lookup\_index = Helper::meta\_data.size() - 1;

                std::vector<float> plato\_confirmed;

                std::vector<float> ending\_fitnesses;

                for (int i = 0; i < Helper::meta\_data.size(); i++) {

                    plato\_confirmed.emplace\_back(Helper::meta\_data[i].plato\_confirmed);

                }

                for (int i = 0; i < Helper::meta\_data.size(); i++){

                    ending\_fitnesses.emplace\_back(Helper::meta\_data[i].ending\_fitness\_height);

                }

                ImPlot::SetNextMarkerStyle(ImPlotMarker\_Circle);

                ImPlot::PlotStems("Plato Confirmed", plato\_confirmed.data(), Helper::meta\_data.size());

                ImPlot::SetNextMarkerStyle(ImPlotMarker\_Circle);

                ImPlot::PlotStems("Ending Fitnesses", ending\_fitnesses.data(), Helper::meta\_data.size());

            }

            ImPlot::EndPlot();

            break;

        }

        case 4: {

            ImGui::BeginChild("Tabled Data", ImVec2(ImGui::GetContentRegionAvail().x, ImGui::GetContentRegionAvail().y), true);

            {

                ImGui::BeginChild("#topfitness", ImVec2(ImGui::GetContentRegionAvail().x / 2, ImGui::GetContentRegionAvail().y), true);

                {

                    ImGui::Text("Top %d Fitnesses", NUMBER\_OF\_RESULTS);

                    for (int i = 0; i < NUMBER\_OF\_RESULTS; i++) {

                        ImGui::SeparatorEx(ImGuiSeparatorFlags\_Horizontal);

                        ImGui::Text("Test %d : placed %d", Helper::Top10Gens()[i].generation, i);

                        ImGui::Text("Fitness: %.10f", Helper::Top10Gens()[i].ending\_fitness\_height);

                        ImGui::Text("Mutation Rate: %f", Helper::Top10Gens()[i].mutation\_rate);

                        ImGui::Text("Mutation Height: %f", Helper::Top10Gens()[i].mutation\_height);

                        ImGui::Text("Tournament Size: %f", Helper::Top10Gens()[i].tournament\_size);

                        ImGui::Text("Solution Fitness: %.10f", Helper::Top10Gens()[i].solution\_fitness);

                        #ifdef PLATO\_HEIGHT

                            ImGui::Text("Plato Confirmed: %i", Helper::Top10Gens()[i].plato\_confirmed);

                        #endif

                        ImGui::SeparatorEx(ImGuiSeparatorFlags\_Horizontal);

                    }

                }

                ImGui::EndChild();

                ImGui::SameLine();

                ImGui::BeginChild("#topsols", ImVec2(ImGui::GetContentRegionAvail().x, ImGui::GetContentRegionAvail().y), true);

                {

                    ImGui::Text("Top %d Solutions", NUMBER\_OF\_RESULTS);

                    for (int i = 0; i < NUMBER\_OF\_RESULTS; i++){

                        ImGui::SeparatorEx(ImGuiSeparatorFlags\_Horizontal);

                        ImGui::Text("Test %d : placed %d", Helper::GetTopSolutions()[i].generation, i);

                        ImGui::Text("Fitness: %.10f", Helper::GetTopSolutions()[i].ending\_fitness\_height);

                        ImGui::Text("Mutation Rate: %f", Helper::GetTopSolutions()[i].mutation\_rate);

                        ImGui::Text("Mutation Height: %f", Helper::GetTopSolutions()[i].mutation\_height);

                        ImGui::Text("Tournament Size: %f", Helper::GetTopSolutions()[i].tournament\_size);

                        ImGui::Text("Solution Fitness: %.10f", Helper::GetTopSolutions()[i].solution\_fitness);

                        #ifdef PLATO\_HEIGHT

                            ImGui::Text("Plato Confirmed: %i", Helper::GetTopSolutions()[i].plato\_confirmed);

                        #endif

                        ImGui::SeparatorEx(ImGuiSeparatorFlags\_Horizontal);

                    }

                }

                ImGui::EndChild();

            }

            ImGui::EndChild();

            break;

        }

        case 5: {

            ImPlot::BeginPlot("Solution Fitness");

            {

                lookup\_index = ImPlot::GetPlotMousePos().x;

                if (lookup\_index < 0)

                    lookup\_index = 0;

                if (lookup\_index > Helper::meta\_data.size() - 1)

                    lookup\_index = Helper::meta\_data.size() - 1;

                std::vector<float> solution\_fitnesses;

                std::vector<float> ending\_fitnesses;

                for (int i = 0; i < Helper::meta\_data.size(); i++) {

                    solution\_fitnesses.emplace\_back(Helper::meta\_data[i].solution\_fitness);

                    ending\_fitnesses.emplace\_back(Helper::meta\_data[i].ending\_fitness\_height);

                }

                ImPlot::PlotLine("Solution Fitness", solution\_fitnesses.data(), Helper::meta\_data.size());

                ImPlot::PlotLine("Ending Fitnesses", ending\_fitnesses.data(), Helper::meta\_data.size());

            }

            ImPlot::EndPlot();

            break;

        }

        default:

            break;

        }

        ImGui::BeginChild("MetaData Lookup", ImVec2(ImGui::GetContentRegionAvail().x, ImGui::GetContentRegionAvail().y), true);

        {

            ImGui::Text("Generation Lookup: %i", lookup\_index);

            ImGui::Text("Mutation Rate: %f", Helper::meta\_data[lookup\_index].mutation\_rate);

            ImGui::Text("Mutation Height: %f", Helper::meta\_data[lookup\_index].mutation\_height);

            ImGui::Text("Tournament Size: %f", Helper::meta\_data[lookup\_index].tournament\_size);

            ImGui::Text("Ending Fitness Height: %.10f", Helper::meta\_data[lookup\_index].ending\_fitness\_height);

            ImGui::Text("Solution Fitness: %.10f", Helper::meta\_data[lookup\_index].solution\_fitness);

            #ifdef PLATO\_HEIGHT

                ImGui::Text("Plato Confirmed: %i", Helper::meta\_data[lookup\_index].plato\_confirmed);

            #endif

        }

        ImGui::EndChild();

    #endif

#endif

}

void gui::Render() noexcept

{

    ImGui::SetNextWindowPos({ 0,0 });

    ImGui::SetNextWindowSize({ WIDTH, HEIGHT });

    ImGui::Begin(

        "AI Assignment",

        0,

        ImGuiWindowFlags\_NoSavedSettings |

        ImGuiWindowFlags\_NoCollapse |

        ImGuiWindowFlags\_NoResize

    );

    {

        gui::Theme();

        {

            gui::Menu();

        }

    }

    ImGui::End();

}

bool gui::graphing() noexcept

{

    CreateHWindow("AI Assignment", "AI Assignment Window");

    if (!CreateDevice())

    {

        DestroyDevice();

        DestroyHWindow();

        return false;

    }

    CreateImGui();

    while (!(GetAsyncKeyState(VK\_END) & 1))

    {

        BeginRender();

        {

            Render();

        }

        EndRender();

    }

    DestroyImGui();

    DestroyDevice();

    DestroyHWindow();

    return true;

}