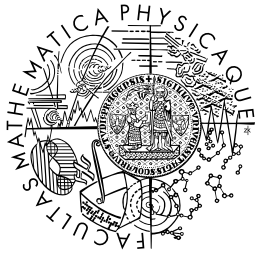

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Marek Nagy

Methods of genetic programming for classification

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Roman Neruda, CSc.

Study programme: Computer Science

Study branch: Artificial Intelligence

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank all those who are related to this project.

First of all, I want to thank my supervisor, Mgr. Roman Neruda CSc. under whose guidance I learned a lot while working on this project. His directions and suggestions have helped in the completion of this thesis.

Finally, I would like to thank my parents, girlfriend and friends who have helped me with their valuable suggestions and guidance and have been very helpful in various stages of project completion.

Title: Methods of genetic programming for classification

Author: Marek Nagy

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract:

This thesis examines using different genetic programming encodings and analyses if they can be used for classification machine learning tasks. We introduce Evolutionary Algorithms and their basic concepts as well as define our specific branch Genetic Programming that is used to generate formulas instead of a differently encoded parametric solution. We introduce Cartesian and Tree-Based encodings and operations on them needed for the algorithm to function properly. The proposed algorithms are implemented and their performance is tested and their results compared on multiple datasets. We then describe how to build and run our solution and discuss the results of the experiments.

Keywords: evolutionary algorithms genetic programming classification

Contents

Introduction

In our day-to-day life everyone has a few challenges to solve: what is the best order of taking partner to work and children to school and what is the best possible price of tomatoes when grocery shopping. Although these problems may sound easy to resolve, at bigger scales of multi-national companies and/or countries they become much harder to solve due to the sheer number of possible solutions. At such extreme sizes we need *optimization* algorithms to help us find the solution.

However, finding the very best result may not be worth the effort for many companies. The reasons may include algorithm uses too much storage or algorithm is too complex (the tools have to be programmed first), but most often the reason is the computation is slow or too slow to be worth the time required. This is where *approximate algorithms* come in. They allow for "good enough" result without worrying about the fact that there may be a better one yet to be found. Approximate algorithms have to be implemented specifically for a given problem which can still be very expensive for the company. This is exactly the reason behind existence of *evolutionary algorithms* (EAs) which while still being problem-specific are much easier to define. They rely heavily on probability of choosing a better solution more often and further modifying it to hopefully achieve a better solution. While EAs generally try to find the best solution and end there, there is also a subclass of EAs, *genetic programming*, that specializes in evolving programs in some kind of formalism.

We live in an era where artificial intelligence (AI) is at every corner: in smartphones, in laptops, personal computers (PCs) and in many web browser-based services: from Google search, Tesla's intelligent electric cars, OpenAI's ChatGPT website to online version of Microsoft's Office suite. The more products use AI the more there is a question about *how* these system actually work. For the most part we have a general intuition of how the machine learning (ML) model is thinking, but this is still only intuition nonetheless. For the majority of people AI is just a black box equivalent to *mostly* reliable magic.

The main goal of the paper is to implement and run experiments of using genetic programming as a supervised machine learning model for classification and generate equations that the models have learned so that we have a better chance to analyse the model and how it works "behind the curtain". Also, a by product of having these equations is that we can only run the equations instead of the whole model (for example neural network) and this way reducing computational power once we want to use them.

In the first chapter we introduce theory behind our paper: evolutionary algorithms, genetic programming and classification. In the second chapter we acquaint ourselves with different encodings of programs and ideas behind them. In the third chapter, we offer our solution proposal including fitness function. In the fourth chapter we provide instructions on how to both build and run the software tool with documentation of all command-line arguments included. In the fifth chapter we design, run and evaluate the experiments to determine if using genetic programming as a machine learning model for classification can be a feasible option in practice.

1. Evolutionary algorithms and genetic programming

Evolutionary algorithms (further marked as EAs) is a group of optimization algorithms used to search for best-possible solutions to a problem that is generally to acquire a solution to, but it is hard to find the best solution (quite a few NP-hard problems belong here). These algorithms were first mentioned by John Holland, his colleagues and students at the University of Michigan. Holland [1992]

They closely mimic process of natural selection, also known as evolution. To be exact, they use several ideas: population of encoded solutions, selection of parents, reproduction (crossover) and mutating individuals and combination of populations. EAs are used in many fields: optimization, automatic programming, immune systems just to name a few. It is usual that the result is not the best possible solution, but that the approximate solution is good enough.

1.1 Basic terminology

Let us first define the terms we need in order to communicate effectively in this thesis.

1.1.1 Chromosome

Representation of a individual is called a *chromosome*. Encoding typically varies from problem to problem and can affect how effective and fast the EA is.

Holland's original proposal suggests that the only encoding should be *binary string*. While technically possible, this is not recommended anymore since more complicated problems require different encoding type for mutations, crossovers and fitness computations to be effective on population scale.

The most popular types of encoding for discrete problems are *binary, tree, permutation*. The most popular types of encoding for continuous problems are: *floating-point numbers, integers*.

1.1.2 Fitness

Fitness is a function that gives us the idea of how good the solution is. It can be precise or heuristic, used as a standalone meaningful value or just for comparison against other solutions in the population.

General rule is that the higher the fitness the better the solution is, but it can be changed to minimization for more human-understandable fitness value.

It is also critical that this function does not take too long to evaluate, since it will typically be computed thousands of times during single run of EA.

1.1.3 Selection

The process of selecting parents for next generation is called *selection*. Basic idea is that the better the individual is the more often it should be chosen, *but*

worse individuals still have to have a chance of being selected. This mechanism is important for avoiding convergence to the local optima.

Selection function is highly dependent on the fitness function and its properties. For example: when it is hard to find numerical fitness function, selection can be a comparison of solutions in a "duel" of sorts. It is also common that one solution can be chosen multiple times thus raising its chance to survive (in a modified version) for more generations.

The most popular a widely-used selections are *roulette-wheel* and *tournament* selections.

Roulette-wheel selection is inspired by a popular game with the same name often found in casinos. The main idea is that the better individuals have larger part of the roulette wheel. It requires that the fitness values of all individuals are non-negative since there is no way to represent negative part of the wheel. As such, it relies on ratio between the fitness values.

Tournament selection is based on tournament spider used in sports. We choose k *uniformly-random* chromosomes from population. Then we construct binary tree with k leafs and compare the pairs. The better node continues to the next "match" and this continues until the top node where the best chromosome will win. In reality, most of the time we simply choose individual with the highest fitness from the chosen solutions.

1.1.4 Crossover

Generating new solution (child) from 2 (or more) previous solutions (parents) is called a *crossover*. It is common that from 2 parents we create 2 children in a single function. Both offsprings contain features from both parents. For example, we will use binary-encoded chromosome with fixed length. The most popular crossovers are: *one-point* and *uniform*. *One-point* crossover is very fast and widely used. Its process can be described as:

1. choose an index that will split both chromosomes into 2 parts
2. take the first part of chromosome 1 and add the second part from chromosome 2 making the first child chromosome
3. take the first part of chromosome 2 and add the second part from chromosome 1 making the second child chromosome

1.1.5 Mutation

Randomly changing a small part of a chromosome is called *mutation*. This part imitates the same named process in nature where something causes a small change of changing DNA a therefore change individuals and their offsprings.

The simplest mutation on binary string is called *bit-flip* since binary string can also be thought of as an array of bits in informatics. By changing one uniformly-randomly chosen digit to the opposite one (thus *flipping* it) we change the chromosome in the least way possible, but this may have big enough impact on the fitness.

The second most popular is called *swap*. As the name suggests, we choose 2 indices and swap the values stored on these indices in the chromosome (binary string).

1.1.6 Environmental Selection

Environmental selection is process of combining 2 consecutive populations. There are 2 most popularones, denoted as: $(\mu + \lambda)$ and (μ, λ). μ represents the size of a population and λ signifies amount of candidates passed to the next generation.

$(\mu + \lambda)$ focuses on combining both populations and choosing the best individuals of size of population since we want the size to stay constant in most cases. It generally converges faster compared to other strategies, but it is also quite probable that it will get stuck in local optima and so not getting the best result possible.

(μ, λ) always chooses the newer population. This means that the fitness is not guaranteed to improve, but the *selection* ensures that it should not be the case too often.

There is also a strategy we can use with any environmental selection strategy called *elitism*. This method ensures that no matter the strategy chosen the k *best* individuals from the previous population survive. This also means that the best solution cannot get worse over time.

1.2 Basic Genetic algorithm

Since we talked about necessary terms, we combine them together into an algorithm representing a basic version of EA, the so-called Simple Genetic algorithm. (presented in figure 1.1)

```
   Input: Amount of generations GENAMOUNT, size of population
          POPSIZE, probability of crossover CROSSPROB, probability
          of mutation MUTPROB
   Output: Last population
   /* Create population with random valid solutions                  */
 1 for i ← 1 to POPSIZE do
 2     population_i ⟵ GetAValidSolution()
 3 for g ← 1 to GENAMOUNT do
 4     for i ← 1 to POPSIZE do
           /* Select parents for next generation                     */
 5         parents_i ⟵ Selection(population)
           /* Crossover parents with given probability, else copy
              from previous generation                               */
 6         if random(0,1) < CXPROB then
 7             nextPopulation_i ⟵ Crossover(parents_i)
 8         else
 9             nextpopulation_i ⟵ population_i
           /* Mutate with given probability                          */
10         if random(0,1) < MUTPROB then
11             nextPopulation_i ⟵ Mutate(nextgen_i)
12     population ⟵ EnvSelection(population, nextPopulation)
13 return population
```

Figure 1.1: Simple Genetic algorithm example

1.3 Genetic programming

Genetic programming is a branch of evolutionary algorithms that focuses on *creating formulae and/or programs* that can produce an algorithm that given the inputs can return values representing the solution. Therefore, during fitness, instead of evaluating a solution we need to run this algorithm, so the evaluation of the algorithms may take longer.

The most common encoding is tree-based encoding since all formulae are quite simple to represent using tree data structure inspired by syntactic trees from programming languages.

1.3.1 Terminology

Since we are creating a tree we have to differentiate between two types of nodes:

- Terminal nodes.
- Non-terminal nodes.

Terminal nodes represent nodes with no children. In our program these are constants and input nodes, but they can be also random nodes.

Non-terminal nodes, as the name suggests, are node with children. These nodes can represent simple functions like addition or multiplication, but also more complicated function requiring some numbers to work. Examples of complex functions are *power* (x^y), *sin(x)* or *condition* node (if $[x > 0]$ then $[y]$ else $[z]$).

1.3.2 Basic operations used in Genetic programming

Chromosome

Since algorithms in computer science generally follow a tree-like structure, data structure called *tree* is a natural way to represent them.

Crossover

Since trees are specific type of graphs, there are many ways of combining them. We chose one of the easier ones:

1. Choose a node (and so its whole sub-tree) from tree representing chromosome in both parts at random
2. Swap the chosen sub-trees between trees

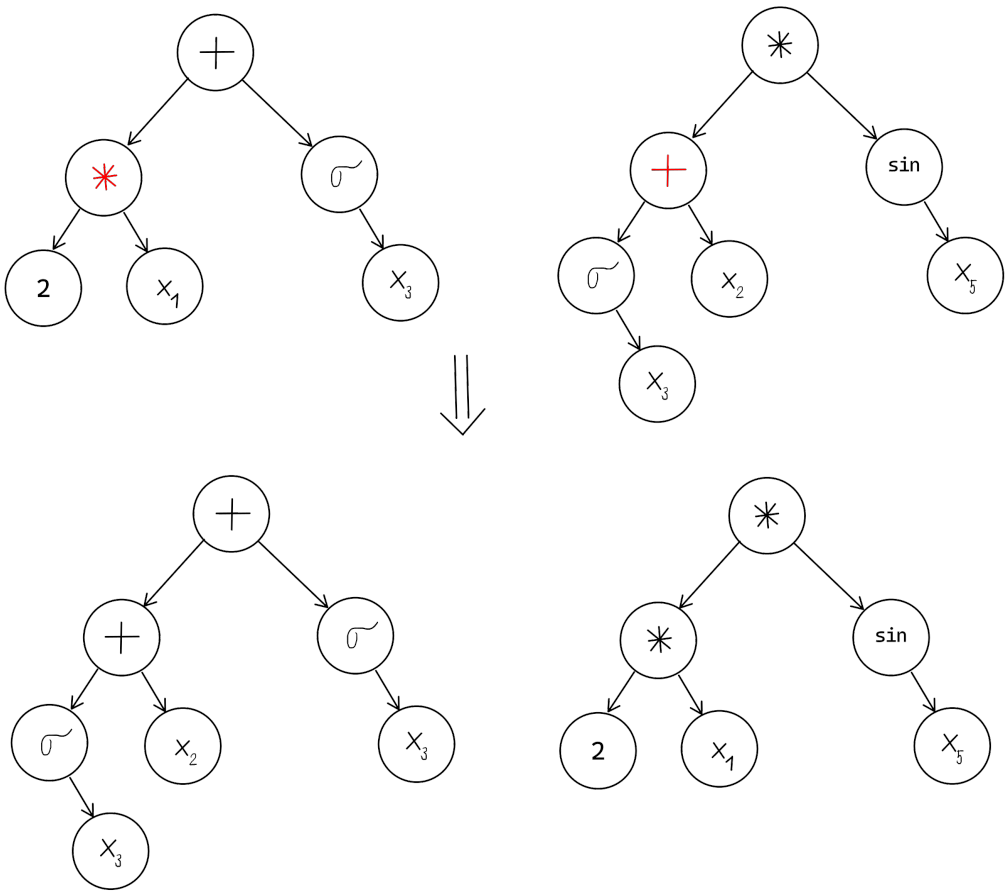

Figure 1.2: Example of crossover

Mutation

We have implemented 2 different mutations:

1. Node change

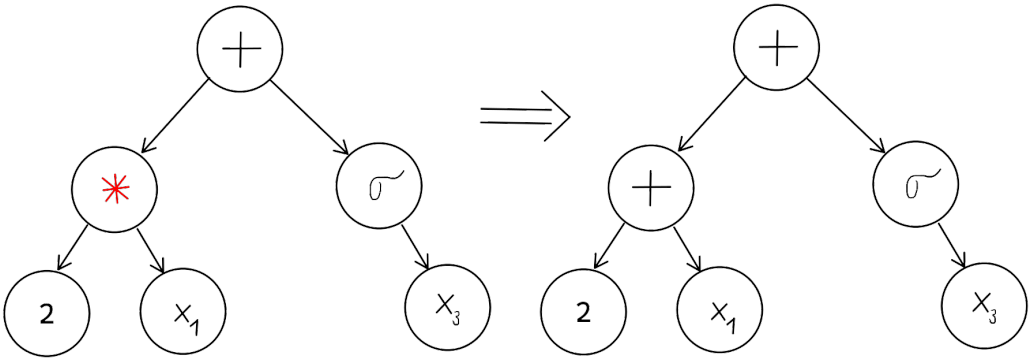

Figure 1.3: Example of *Node Change* mutation

2. Children shuffle

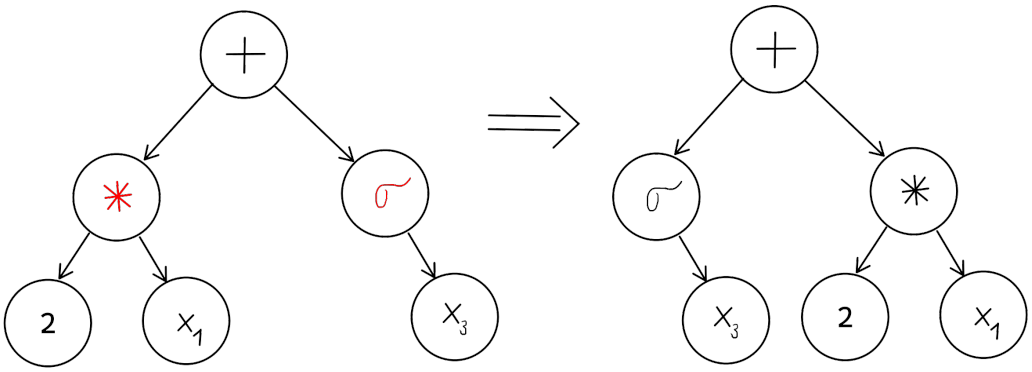

Figure 1.4: Example of *Children Shuffle* mutation

Node change chooses one node uniformly randomly and then changes its functionality, for example from addition to multiplication. Each functionality has configurable weight so we can also control the ratio of different functionalities used.

Children shuffle randomly chooses a node with children and changes their order and so different sub-trees can end up in different part of the multi variable function.

Fitness

To compute the fitness of the individual is very straightforward: for each test case we change input nodes' values according to the test case (list of input values) and run the algorithm. This can be sped up by using multi-threading, but is still considered quite slow compared to other EA types' fitness evaluation, although some other methods of machine learning (neural networks, reinforcement learning) can run even slower.

1.4 Classification

Since we apply genetic programming on classification tasks, let us first define what classification is. *Classification* is one of the most widely known tasks for machine learning (ML): regression and classification.

Regression is used as a way of computing a value/values on output using different ML methods. Most common models used for regression task are linear regression, neural networks. There are quite a few ways of evaluating performance of a model, but the basic function used is *mean squared error* usually denoted as *MSE*:

$$MSE_{model} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(model.predict(x_i) - y_i)^2}$$

where N represents amount of input patterns, y_i represents i^{th} result and x_i represents i^{th} input.

Classification is used to categorize input patterns into predefined categories. It is also possible (and used) to instead of prediction to get probability of input belonging to each of the categories. There is plenty of ways of evaluating classification models, but the simplest one is *accuracy* defined as

$$accuracy_{model} = \frac{\sum_{i=1}^{N} compare(model.predict(x_i), y_i)}{N}$$

$$compare(z, y) = 1 \; if \; z = y \; else \; compare(z, y) = 0$$

where N represents amount of input patterns, y_i represents i^{th} result and x_i represents i^{th} input. Value of model's accuracy is always in interval $[0, 1]$. There are also more complicated method of evaluation, but since we are running this method hundreds to thousands of times per each run of the algorithm we decided to use the simplest one.

2. Different encodings for Genetic programming

In this thesis, we are trying to use genetic programming as classification model generator so we are using 2 most known GP encodings: *tree-based* and *cartesian*.

2.1 Cartesian encoding

Cartesian encoding uses idea that we can arrange nodes in *cartesian coordinate space* or *a 2D grid* and then let the nodes themselves choose a parent. This way we can have multiple outputs with all having chance to use the same mid-level results and by that avoiding re-evolving them and speeding up the evolution in theory.

This type of GP encoding was discovered independently by multiple people: Sushil Louis and R. Poli. "Louis described a binary genotype that encodes a network of digital logic gates, in which gates in each column can be connected to the gates in the previous column." Miller [2011] "R. Poli, inspired by neural networks, proposed a graph-based form of GP called parallel distributed GP (PDGP)." Miller [2011]

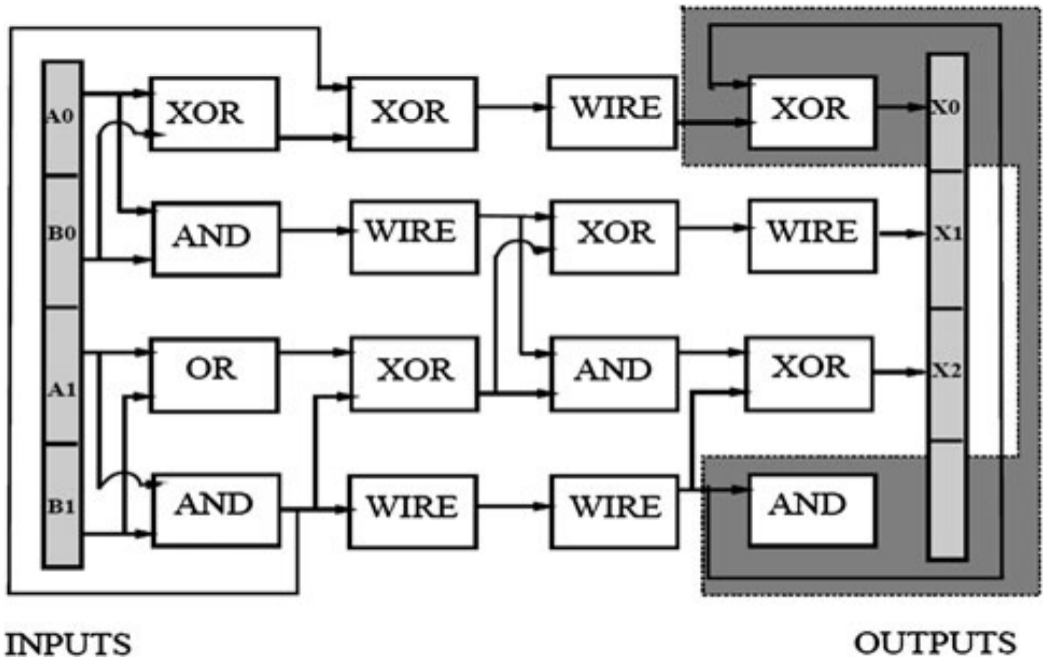

Figure 2.1: Louis' genotype representation from Miller [2011]

There are two definitions on what a Cartesian GP (further CGP) exactly is. Both definitions state that it is a way of evolving a graph structure and thus being able to evolve more complex systems than its tree counterpart. It uses the idea of

placing nodes on a grid and connecting them in a layered manner Miller [2011]. We implement an iteration on this idea that can dynamically change amount of layers of nodes and number of nodes in each layer.

2.1.1 Chromosome

In our implementation we can imagine the following: our chromosome consists of layers of nodes where each node is connected to several *parent nodes*. The node can (but does not have to) use any number of these parents. Each of these parent nodes has to be from one of the previous layers while not all parent nodes have to be from the same previous layer. Layers themselves are also *not the same size* during the run of the GP algorithm. Every node has layer index. The first layer is made up of input nodes. Every other node has a reference to functionality class so that we can change behaviour of the node without creating new instance. We have several types of nodes where each type refers to a specific operation. Every node type also has an associated weight (later normalized for probability) for easier way of preferring a specific node.

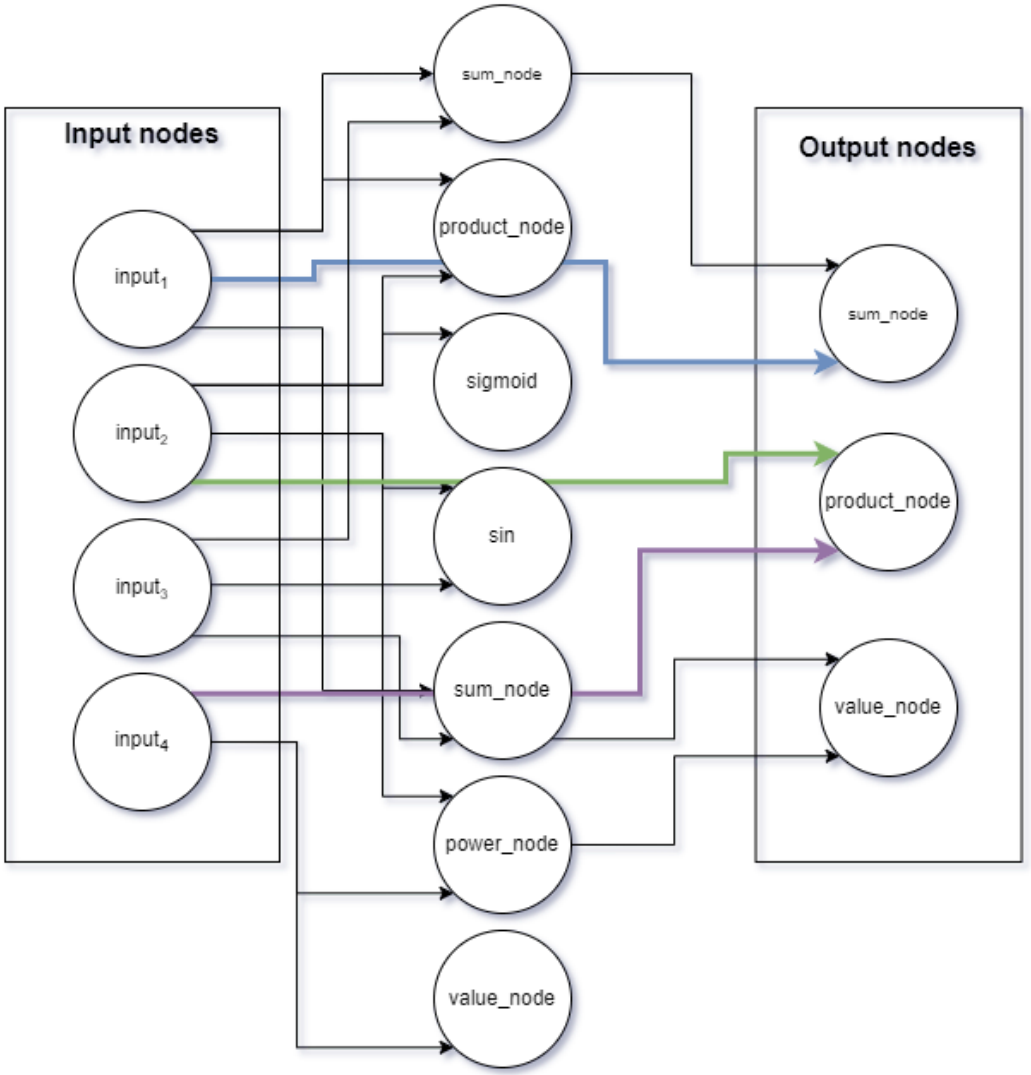

Figure 2.2: CGP chromosome example (coloured paths go across 1+ layers)

2.1.2 Fitness

For *fitness function*, we are using accuracy calculated on predictions in one-hot encoding (calculated in $(1 - accuracy)$ to maintain "lower is better" approach). Calculating predictions *can* be done in parallel, but since we don't use very large datasets we did *not* implement parallel computation in order to simplify the code and increase its readability.

2.1.3 Selection

Our choice for selection is *tournament selection*. We pick uniformly-randomly K individuals from the last population, skip trying to build a tournament spider and we simply pick the individual with the best fitness.

2.1.4 Crossover

For crossover, we went with what we call a *fixed index crossover*.

Since we are crossing individuals of potentially different amount of layers ("chromosome's length" further) we find minimum of the lengths of the individuals and randomly choose an index i from interval $[1, i] \cap \mathbb{N}$. Afterwards, we split layers into 2 parts and exchange those parts between chromosomes like in one-point crossover. Then we fix possible errors in connections that might have happened.

It is possible that the node refers to i^{th} (parent) node in j^{th} layer but in the new chromosome $|layers_j| < i$ and therefore we would get an index error if not corrected. We fix this by referring the node to last node in j^th layer.

2.1.5 Mutation

We implement multiple mutations and the EA library ensures that *each of the mutations* has a probability of being run *for each individual*.

Change node mutation

In this mutation, we change the node type in order to try to achieve better accuracy. New type of node is chosen randomly according to the weights specified using command line arguments.

Add layer mutation

We choose an index of a layer between inputs and outputs and insert a new layer into this index. For all nodes we choose their parents randomly from previous layers. All nodes' parents' indices from new layer to outputs are fixed after the insert.

Remove layer mutation

This mutation removes a random layer (excluding input and output layer) from the chromosome and restores indices to correct ones if possible, otherwise chooses new random parent for node.

Change parents mutation

This mutation has an internal probability of changing parents of *each node* on the chromosome *except* the first (inputs) layer of nodes. Each parent is chosen using uniform random choice of a layer (from layers prior to the layer of the node we are changing the parents of) and then a random uniform choice of a node in the chosen layer.

Add node from layer mutation

This mutation chooses a random layer of nodes (excluding input and output layer) and adds a new random node based on weights of each node type. Parents are chosen the same way as in *Change parents mutation* mentioned above in 2.1.5.

Remove node from layer mutation

This mutation uniformly randomly chooses a layer of nodes (excluding input and output layer) and then also uniformly randomly chooses a node to remove. Then it checks if any nodes from latter node layers pointed to the removed node (if yes, we choose new parent) and if they pointed to the nodes "above" the removed node int he same layer (if yes, we fix node index).

2.2 Tree-based encoding

Tree-based encoding focuses on encoding presented by John Koza (Koza [1992]) and it is most widely known for genetic programming's purposes.

2.2.1 Chromosome

Basic structure represents a tree where tree is a specific kind of graph from discrete mathematics. This represents entry point into the structure and most functions (both mathematical and programmatic) are called on this node and computed recursively on the whole structure.

Chromosome is implemented in the following way: we create an output tree for each of the output classes and it returns a real number. The main idea being *"the higher the number the more likely it is we should use the given output class"* without restricting the output into interval $[0, 1]$. Then we can use *softmax* function to convert these numbers to probabilities. We also implemented a tertiary tree since we want to use functions with at most 3 children required.

2.2.2 Fitness

For fitness we choose a modified accuracy score calculated on *train set*. The modification consists of adding formula $\frac{2*individual.Depth()}{|inputs|}$ representing soft restriction on depth of trees. So the final formula is

$$\min_{\forall ind \in population} (1 - ind.accuracy) + \frac{2 * ind.depth}{|inputs|}$$

where the first part $(1 - accuracy)$ will be referred to as *score*. We do this so we have a more describable number to watch and understand how the algorithm evolves through time.

We choose the class of which respective tree generated the highest number. We can first calculate probabilities using *softmax* but since it is monotone and rising function choosing max before and after using softmax does not change the outcome.

2.2.3 Selection

For selection we choose tournament selection since we want to choose the better individuals and don't regard the ratio of the fitness values.

2.2.4 Crossover

We choose to implement a *sub-tree swap* crossover on each of the trees. We take the 2 trees representing class number 1 (one from each chromosome), pick a random node with children, take a child node (and with it the whole sub-tree) and swap them. We repeat this process for each output class.

2.2.5 Mutation

In this mutation we change functionality of a node in order to try to achieve better accuracy. New functionality is chosen randomly according to the weights specified using command line arguments.

This mutation can also change terminal node to non-terminal and vice versa. If we are changing terminal to non-terminal node we generate a new sub-trees for children with specified maximal depth. If we are changing non-terminal one to terminal, we remove the children.

For example, if we have a node that is used to sum results of its children, we can swap its functionality for product without changing the structure of the chromosome.

2.3 Nodes' functionalities

We implemented several functions for both encodings so here is a brief explanation of each of them. Although in cartesian encoding we refer to nodes that are used as source of values for other nodes as *parents*, in this part we are calling them *children* just to unify terminology of encodings and to not repeat ourselves.

2.3.1 Non-terminal

Non-terminal functionalities require 1 or more values from children nodes to function properly.

Sin

We take the first child's result and apply *sin* function to it.

$$sin\_node.result = sin(child1.result)$$

ReLU

We take the first child's result and apply $ReLU$ function to it.

```
1 if child1.result > 0 then
2     ReLU_node.result = child1.result
3 else
4     ReLU_node.result = 0
```

Sigmoid

We take the first child's result and apply *sigmoid* function to it.

Sigmoid function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

so

$$sigmoid\_node.result = \sigma(child1.result) = \frac{1}{1 + e^{-child1.result}}$$

Unary minus

We take the first child's result and multiply it by *-1*.

$$unary\_minus\_node.result = -child1.result$$

Sum

We add the results of node's 2 children together.

$$sum\_node.result = child1.result + child2.result$$

Product

We multiply the results of node's 2 children together.

$$product\_node.result = child1.result * child2.result$$

Power

We take result of the first child and raise it to the second child's result.

$$power\_node.result = child1.result^{(child2.result)}$$

This also has its problems: 0^0 or $(-1)^{\frac{1}{2}} = \sqrt{-1}$. In both of these cases, we set result to infinity. Since this rule follows our fitness and intuition behind it (lower fitness means better individual), the resulting fitness of the chromosome will be infinity and therefore will be the worst candidate in population. Only way a chromosome with infinite fitness can be chosen is if in tournament selection all chosen nodes have infinite fitness which we assume is extremely unlikely.

Condition

This represents a simple *if ... then ... else ...* concept from computer science. We compute its result in the following way:

```
1 if child1.result > 0 then
2     condition_node.result = child2.result
3 else
4     condition_node.result = child3.result
```

2.3.2 Terminal

Terminal functionalities can be as function that returns value with no input value or unrelated to the input, such as constants or input values.

Value

We fix value of node during its creation. The value nodes are specified in the beginning of EA and the values do not change and we do not add more value nodes during algorithm.

Input

This is a special type of value node that changes value according to the input when computing result for it. It can also remember which index of input it references and when getting representation, it prints it out in form x_{index}.

3. Our solution proposal

Since our task is highly dependent on the performance of our code we decided against the "industry standard" programming language Python in favour of C# and .NET platform. This way we can use multi-threading more effectively which is severely limited in Python due to *global interpreted lock* (GIL). In addition, C# is compiled into *intermediate language* (further IL) [Microsoft [a]] and having static type system means the runtime or compiler can make the program run faster than in Python which is an interpreted language.

In order to reuse as much code as possible, we implemented a simple framework for running general genetic algorithms, but can be used for *GP* as well. It uses many advanced concepts and techniques including generics, thread-pools and LINQ that help us to create more robust and performant but still very flexible framework.

We have created multiple projects to keep functionality separated and we can use command *'dotnet build'* to build all target projects including the ones they depend on (also called *dependencies*). We defined a dependency graph (figure 3.1) *excluding external dependencies* that is easier to read and faster to understand than looking at the dotnet solution files.

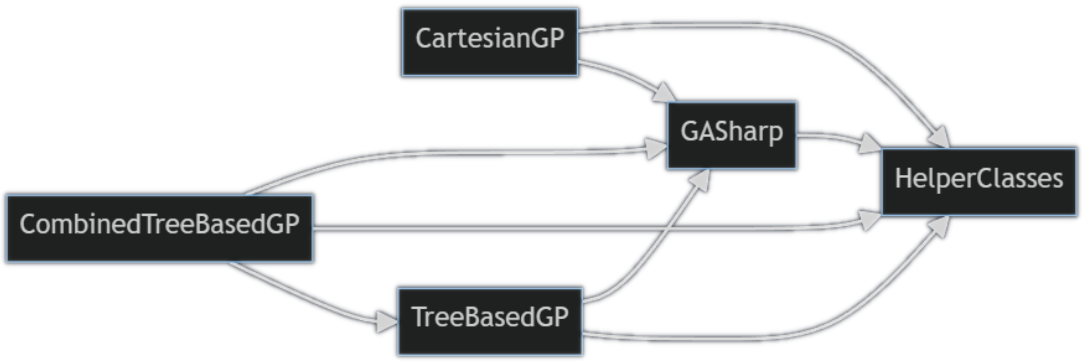

Figure 3.1: C# project dependency graph

HelperClasses is a simple library which includes classes and methods used in other projects from this solution. *GASharp* is a library that includes important generic classes for running the GA in general (more details in the following section). *CartesianGP* is the project where we create executable for GP using *cartesian* encoding. *TreeBasedGP* is the project where we implement GP using *tree-based* encoding. This is meant as stepping stone for the following project and

not to be run. In this project, each chromosome is only one tree. *CombinedTree-BasedGP* is the project where we create executable for GP using an ensemble of chromosomes with *tree-based* encoding.

3.1 Shared library (GASharp)

We implemented a framework based on algorithm described in section 1.2 for easier prototyping and to ensure the basis of algorithm stays the same in both our encodings. It also includes basic templates for classes:

- Chromosome

```
1  public abstract class Chromosome<T> where T: Chromosome<T>
2  {
3      public double Fitness { get; set; }
4      public abstract T CreateNew();
5      public abstract T Clone(bool preserveFitness = false);
6      public virtual void UpdateFitness(Fitness<T> fitness)
7      {
8          this.Fitness = fitness.ComputeFitness((T)this);
9      }
10     public abstract bool IsValid();
11 }
```

Figure 3.2: Base Chromosome class

- Crossover

```
1  public abstract class Crossover<T>
2  {
3      public double CrossoverProbability { get; protected set; }
4      public abstract Tuple<T, T> Cross(T ind1, T ind2);
5  }
```

Figure 3.3: Base Crossover class

- Mutation

```
public abstract class Mutation<T>
{
    public Mutation(double probability)
    {
        if (probability < 0d || probability > 1d)
            throw new ArgumentOutOfRangeException(
                $"Probability is expected from interval [0,1]."
                + "Received {probability}"
            );

        this.MutationProbability = probability;
    }
    /// <summary>
    /// Probability that the mutation will occur.
    /// </summary>
    public double MutationProbability { get; private set; }
    public abstract T Mutate(T ind, int genNum);
}
```

Figure 3.4: Base Mutation class

- Fitness

```
abstract public class Fitness<T> where T: Chromosome<T>
{
    public abstract double ComputeFitness(T ind);
    /// <summary>
    /// Thread-safe implementation of computing fitness for whole population.
    /// </summary>
    public virtual void ComputeFitnessPopulation(T[] population)
    {
        foreach (var ind in population)
        {
            this.ComputeFitness(ind);
        }
    }
}
```

Figure 3.5: Base Fitness class

- Population Combination Strategy

```
1  public abstract class PopulationCombinationStrategy<T>
2  {
3      public abstract T[] Combine(T[] oldPopulation, T[] newPopulation);
4  }
```

Figure 3.6: Base PopulationCombinationStrategy class

- Selection

```
1  public abstract class Selection<T>
2  {
3      public abstract Tuple<T, T> ChooseParents(IReadOnlyList<T> population);
4      /// <summary>
5      /// Prevent having to recalculate probabilities by already providing them.
6      /// </summary>
7      public abstract Tuple<T, T> ChooseParents(
8          IReadOnlyList<T> population,
9          IReadOnlyList<double> probabilities
10     );
11 }
```

Figure 3.7: Base Selection class

GASharp, this library, utilizes *generic programming* so the developer using this library is not constrained by predefined methods and can fully use the custom classes defined. Prefix *Max-* means it can be used in GA that *maximizes* fitness whereas *Min-* is used for GA that *minimizes* fitness. It contains a few most used *selections*:

- MinTournament
- MaxTournament
- MinRouletteWheel
- MaxRouletteWheel

and *population combinations*:

- Take New (also known as (μ, λ))
- MinElitism (with given number of elites)

- MaxElitism (with given number of elites)
- MinCombineBest (also known as $(\mu + \lambda)$)
- MaxCombineBest (also known as $(\mu + \lambda)$).

Min- and MaxElitism also include argument for creating a given amount of new individuals to preserve diversity of the population during the run of the algorithm.

It supports setting additional parameters via public properties (these are set from *command line arguments* or *json* file in concrete GPs):

- MinThreads (mapped into .NET's ThreadPool corresponding method)
- MaxThreads (mapped into .NET's ThreadPool corresponding method)
- MaxGenerations
- PopulationSize
- CrossoverProbability
- MutationProbability

However, the meaning of *MutationProbability* and *CrossoverProbability* is different in our framework. While the MutationProbability signifies probability for each of the defined mutations to occur, the CrossoverProbability denotes **if** a crossover is applied and if it is, uniformly randomly choose a crossover class. This way we ensure we use at most one crossover, but we may use multiple mutations.

3.1.1 Creating GA class

Since the GA class is generic and takes quite a few parameters and argument, this section is made as a guide to creating it.

```
1  public GeneticAlgorithm(Func<T> createNewFunc,
2          IList<Mutation<T>> mutations,
3          Crossover<T>[] crossovers,
4          Fitness<T> fitness,
5          Selection<T> selection,
6          PopulationCombinationStrategy<T> popCombination)
```

Figure 3.8: Code snippet of GA constructor

We will go through each argument one-by-one and explain what each argument does and when and how it is used.

Func<T> createNewFunc

As the name implies, this argument is used for creating a new individual of class T representing the *chromosome class*. It is needed mainly for creating a new population in the beginning, but can be also used in Min- and MaxElitism population combination strategy to preserve diversity throughout the run of the GA.

IList<Mutation<T>> mutations

This argument represents a list of initialized mutation classes' instances. The *IList* inteface is used so that this variable *can* be altered during the algorithm is already running.

Each mutation has a *MutationProbability* to be applied for each individual independently of other mutations and has to implement a method *Mutate(T ind, int genNum)* that *returns T* and inherits from the main Mutation<T> class (see Figure 3.4).

Crossover<T>[ ] crossovers

This argument represents ordered list of initialized crossover classes. Each of them has to inherit from main *Crossover<T>* class (see Figure 3.3) and implement method *Cross(T ind1, T ind2)* that returns *Tuple<T, T>*.

Fitness<T> fitness

This argument represents our fitness function to be used in the GA. The main methods used are called *ComputeFitness(T ind)* and *ComputeFitnessPopulation(T[ ] population)*.

ComputeFitness(T ind) is the default implementation of evaluating fitness for a single individual. This implementation should be implemented in all implementations since *single-threaded implementation running of GA is the fallback* (in case of using single thread for both *MinThreads and MaxThreads* parameters) *of the multi-threaded* method.

ComputeFitnessPopulation(T[ ] population) is used in multi-threaded version of GA in order to optimize computation and take advantage of additional threads on the system. The recommended approach is to utilize .NET's *PLINQ* technology.

Selection<T> selection

This argument represents our chosen selection function used to choose parents from previous generation to create a new generation. It has to inherit from main *Selection<T> class* and implement method *ChooseParents(IReadOnlyList<T> population, IReadOnlyList<double> probabilities)* that returns *Tuple<T, T>*.

PopulationCombinationStrategy<T> popCombination

This argument represents class responsible for combining previous population with the next one.

```
var ga = new GeneticAlgorithm<CartesianChromosome>(
            () => CartesianChromosome.CreateNewRandom(
                layerSizes,
                cliArgs.TerminalNodesProbability,
                terminalNodesProbabilities,
                nonTerminalNodesProbabilities
            ),
            [new ChangeNodeMutation(
                cliArgs.PercentageToChange,
                cliArgs.ChangeNodeMutationProbability,
                cliArgs.TerminalNodesProbability,
                nonTerminalNodesProbabilities,
                terminalNodesProbabilities
            )],
            [new FixedIndexCrossover()],
            trainAccuracy,
            new MinTournamentSelection<CartesianChromosome>(folds: 5),
            new TakeNewCombination<CartesianChromosome>()
        );
```

Figure 3.9: Example of initializing GA class from Cartesian GP main function

3.1.2 Other class properties

Our GA class *GeneticAlgorithm* also has a few public properties that can be changed any time after initialization of a variable. These properties include: *CrossoverProbability, MaxGenerations, PopulationSize, MinThreads*, and *MaxThreads*.

MinThreads and MaxThreads

These parameters need to be *positive integers* with additional condition of

$$MinThreads \leq MaxThreads$$

These properties are directly passed to .NET's static *ThreadPool* [Microsoft [c]] class and the algorithm uses *MaxThreads* as *DegreeOfParallelism*[1] internally *via .NET's PLINQ technology* [Microsoft [b]].

MaxGenerations

This parameter represents the maximum amount of generations to run. That is also a way to send a *stop condition* function into method running the GA.

PopulationSize

This parameter represents a size of population throughout the algorithm. This size is *constant* and does not change neither can it change.

[1] https://learn.microsoft.com/en-us/dotnet/api/system.linq.parallelenumerable.withdegreeofparallelism?view
8.0

3.2 Cartesian GP

3.2.1 Chromosome implementation

Our implementation of encoding from section 2.1 focuses on grouping nodes into layers (similarly to neural network Multi-Layer Perceptron [Haykin [1994]], but in this case neurons themselves can choose a parent from *any* of the previous layers of neurons.

In the class itself we distinguish between input layer of nodes and all others. This is to make sure to not change input layer's behaviour or size. Each of the nodes has number reference to its 3 parents where for each parent it stores global layer index and node index within that layer. This means that we can change the parent node without altering the child node's attributes or data.

Each node contains the index references to its parents (excluding input nodes) and implements a *Compute* and *GetRepresentation* functions. **Compute** function is responsible for computing the result and returning it whereas **GetRepresentation** adds representation of the node into a *StringBuilder* class for more effective creating of the representation string.

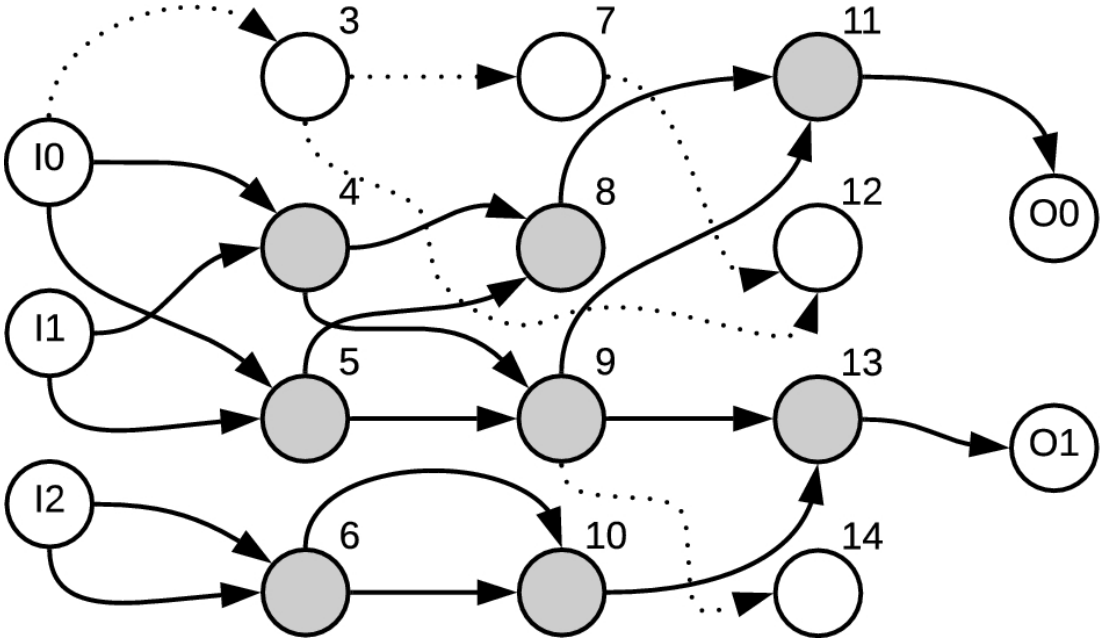

Figure 3.10: Visualization of Cartesian GP chromosome from Spryn

Difference between our implementation and the image above is that *we allow parents to be from more layers than only a previous layer.*

Chromosome also has method for computing prediction on given input. This is achieved via setting input nodes to their respective values and then using DFS (depth-first search) from output nodes to compute the result. Then we compute *one-hot* encoding on the resulting array.

The chromosome contains method to compute probabilities from the values

using the *softmax* function

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

for N inputs. This is inspired by neural network's way of computing probabilities for classification using the aforementioned function [Bridle [1989]]. Although this method is not used in computing fitness, it can be used in production code after expanding our code with methods for saving and loading the instance of the class. We do not implement such methods since the main idea of this thesis is to product resulting formulas and not the instances of the class.

3.2.2 Fitness

Fitness is calculated using accuracy on a train set (specify using CLI argument *--train-csv*). Train set is loaded into memory and is stored there for the whole length of the GA. After running the GA, accuracy is evaluated on a test set and stored in *.txt* file in output directory created by the program. If we do not specify test set, accuracy is calculated on the train set.

If we use multi-threading, we can quite simply parallelize fitness computation of the whole population since input values are set internally for each individual of the population. Our implementation uses *.NET's PLINQ* technology so that the framework can take care of details while we specify flow in functional manner.

We are computing accuracy based on predicted values for each class and then encoding it into *one-hot* encoding where 1 denotes the given class had the highest number. Afterwards, we compare it to the desired output that is also expected to be encoded in *one-hot encoding*. We also added attribute *Score* to chromosome's implementation. Score represents $(1 - accuracy)$ metric while *Fitness* includes both Score and penalty for depth of the chromosome in order to control bloat. Explicitly:

$$\forall ind \in population$$

$$ind.Score = 1 - ind.accuracy$$

$$ind.Fitness = ind.Score + ind.Depth() * \frac{2}{|total\_inputs|}$$

3.2.3 Crossover

Since chromosomes can have different amount of layers of nodes we decided to do a variant of one-point crossover over the smaller of their two depths:

1. Find common depth of the two chromosomes (the smaller of the two)
2. Choose an index for separation
3. Create new chromosomes by switching the parts after index
    - First new chromsome from 1^{st} part of 1^{st} chromosome and 2^{nd} part of 2^{nd} chromosome
    - Second new chromsome from 1^{st} part of 2^{nd} chromosome and 2^{nd} part of 1^{st} chromosome

4. Check indexing and fix if needed

    - Node can have parent from a layer that has more nodes than in a new chromosome. If so, replace with parent on index $previous\_index \ mod \ |new\_layer|$.

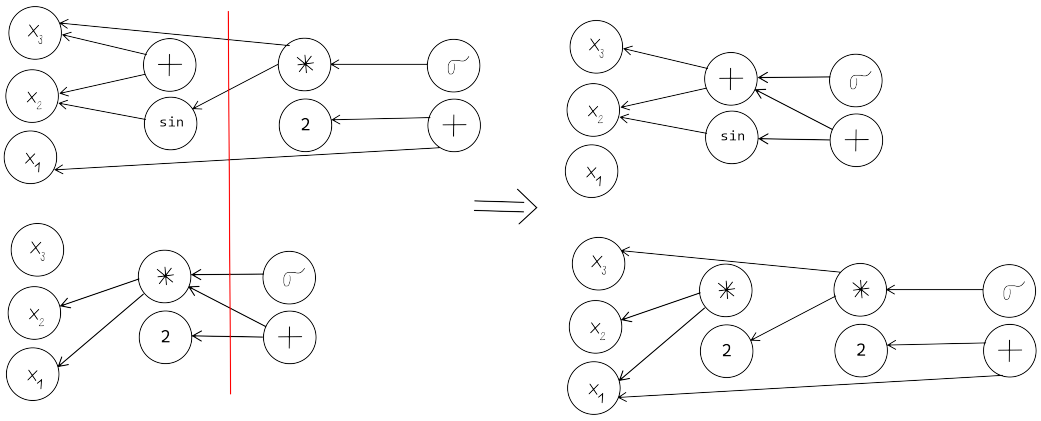

Figure 3.11: Example of crossover chromosomes from Cartesian GP

3.2.4 Mutations

Add node to layer mutation

With a given probability add a node to a uniformly randomly chosen layer and randomly choose its parents.

Add layer mutation

With a given probability add a new layer on uniformly randomly chosen index, then for all nodes from this new layer choose parents randomly. Afterwards fix indices in layers after generated layer so that they reflect the previous state of chromosome.

Remove layer mutation

With a given probability remove a single layer (except input and out layer) from chromosome. Afterwards, fix parents' indices on nodes that are pointed to layers after the removed layer. For nodes that previously pointed to node in the removed layer, randomly choose new parents.

Remove node from layer mutation

With a given probability uniformly randomly choose a layer (excluding input and output layer) and the uniformly randomly choose a node from the chosen layer. If any node pointed to the removed node, uniformly randomly choose a new parent from uniformly random layer and uniformly random node from that layer.

Change node mutation

With a given probability go over all nodes, except input nodes and randomly change *--percentage-to-change* percent of the chromosomes' class and because of it, their functionality. This changes functionality of the node and does not create a new node.

Change parents mutation

With a given probability go over all nodes expect input nodes and randomly change *--percentage-to-change* percent of the chromosomes' parents and because of it, their input nodes. This changes functionality of the node and does not create a new node.

3.3 Tree-based GP

3.3.1 Chromosome implementation

We are implementing encoding from section 2.2 using a *tree* graph structure consisting of nodes that we don't expect to be very deep (explained further), it makes sense to use recursive version of the structure.

We can quite simply compute result of the tree using DFS from root node where children nodes represent potential inputs for the node's function meaning that when computing only nodes on path from root node to the node computing (worst case leaf node) are on the stack and thus we (practically) don't have to worry about problems with too deep recursion.

Technically, trees can grow during the algorithm, but based on our testing we will run out of memory far sooner than hitting the recursion limit.

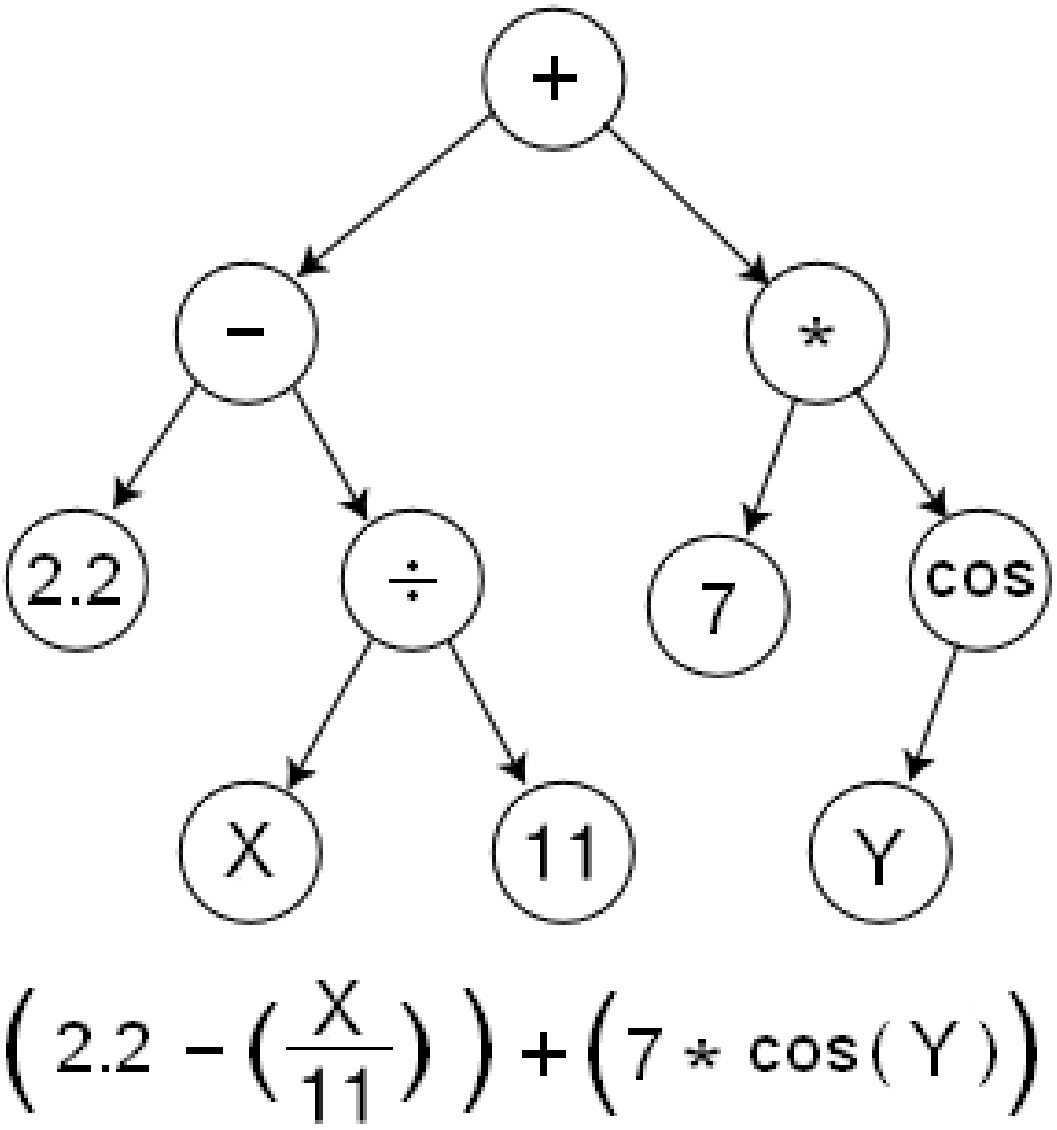

Figure 3.12: Visualization of TreeBased GP chromosome from Spryn with produced formula

The visualization above comes from Spryn, but *this visualization* contains a **few differences**:

- We use tertiary tree instead of binary to allow for use of conditional node
- Our nodes have only either 3 children or none (unary operator still has 3 subtrees)
- In our implementation, we use functionality class (sum, product, condition ...) separated from the node class to avoid creating new nodes and thus reducing usage of memory.

Input functionality class' instances remember their input index and are "global" variables meaning that setting input values and evaluating an individual from population are separate actions that depend on each other.

3.3.2 Fitness details

Since this project is only a stepping stone for the combined tree-based GP, we decided to not describe fitness of the chromosome. For more information, the theory behind computing fitness can be seen in section 2.2.2.

3.4 Combined Tree-based GP

3.4.1 Chromosome implementation

This chromosome represents a special ensemble of TreeBasedGP's chromosomes from section 3.3 where for each output class of classification we build one such chromosome. When computing prediction, we compute *argmax* of results of tree for each output. Explicitly:

$$model.predict(x) = \underset{i \in output\_classes}{\operatorname{argmax}} \; tree_i.GetResult()$$

With regards to how mutation operations are made, we create a class for each of the mutations provided in TreeBasedGP project and use them on each of the trees in the chromosome. Crossover is made similarly: if the GA runs the crossover we apply crossover on each pair of trees (one tree from each chromosome for the same output class) and return the result.

3.4.2 Fitness details

Fitness is calculated using accuracy on a train set (specify using CLI argument *--train-csv*). Train set is loaded into memory and is stored there for the whole length of the GA. After running the GA, accuracy is evaluated on a test set and stored in *.txt* file in output directory created by the program. If we do not specify test set, accuracy is calculated on the train set.

If we use multi-threading, we can quite simply parallelize fitness computation of the whole population by running computation of each individual in separate thread. One difference compared to the CartesianGP is that here *Input nodes are global variables* so that they can be used wherever in the trees, but to also allow for simple and fast change of value when computing fitness and going through many different inputs. Our implementation uses *.NET's PLINQ* technology so that the framework can take care of details while we specify flow in functional manner.

We are computing accuracy based on predicted values for each class and then encoding it into *one-hot* encoding where 1 denotes the given class had the highest number. Afterwards, we compare it to the desired output that is also expected to be encoded in *one-hot encoding*. We also added attribute *Score* to chromosome's implementation. Score represents $(1 - accuracy)$ metric while *Fitness* includes both Score and penalty for depth of the chromosome in order to control bloat. Explicitly:

$$\forall ind \in population$$

$$ind.Score = 1 - ind.accuracy$$

$$ind.Fitness = ind.Score + ind.Depth() * \frac{2}{|total\_inputs|}$$

We implement crossover already mentioned in section 1.3.2, but applied to consecutive pairs (1^{st} tree from 1^{st} chromosome with 1^{st} tree from 2^{nd} chromosome and so on):

1. We find all nodes that have children for both chromosomes
2. We choose one node from each chromosome uniformly randomly
3. We switch their places in their respective chromosomes
    - By changing their place we also switched whole subtrees

3.4.3 Mutations

Change node mutation

We implement mutation number 1 from section 1.3.2 on each of the trees from our combined chromosome. This is implemented for the whole tree using *--percentage-to-change* command-line argument. We implement it in recursive manner since we don't expect the trees to be deep enough to cause problems in recursion. Once the node is selected for mutation, we choose whether we are replacing it with uniformly randomly chosen terminal node or uniformly randomly chosen non-terminal node based on *--terminal-nodes-probability* command-line argument. If non-terminal node was chosen, we also create new subtrees as its children using full tree of depth specified by *--depth* command-line argument.

Shuffle children mutation

We implement mutation number 2 from section 1.3.2 on each tree from our combined chromosome.

4. How to build and run GPs

This chapter includes information how to build and run the main algorithms written in C#. In 4.1, we cover compilation of C# programs and creation and activation of virtual environment for Python. In 4.2, we provide user manual for running the most important Python scripts and the C# programs with explanation for its command-line flags.

4.1 How to build

4.1.1 Requirements

First of all, we need to clone our GitHub online repository from its publicly available URL [1]. This way we actually have the code we need to build the program.

We also need *.NET 8* framework (newer releases might work but it is not guaranteed) installed on our operating system (further OS). This framework is open-source and cross-platform for all major systems: Windows, Mac and all Linux-based OS.

We also included Python scripts for preparing input CSV (comma-separated values, **not neccessary separated by comma**) files generating graphs about algorithm's evolving over time. These are not essential to the algorithm itself, but are of great help if we choose to use them. These use *Python 3.9* so we can download it from Python's official website[2]. We saved *requirements.txt* containing names of installed modules with their versions. Our recommendation is to create new virtual environment and install the modules there. Since we used module *virualenv* for creating the virtual environment we will use it to create the environment, but there are multiple choices for creating it (conda, poetry, installed directly onto the system [**not recommended**]), so if we want to we can choose another tool.

We also expect for all building and publishing to be taking place on **Windows**, on Windows 10 to be exact, although this method should work just fine on Windows 11 as well. We also expect you to have administrator's rights meaning you can install new software on your machine.

4.1.2 Setting up

.NET and C#

After downloading, we navigate to directory *MasterProject*,

[1] https://github.com/marnagy/MastersProject
[2] https://python.org

Figure 4.1: Example of path to MasterProject

and run prepared script (*build_win_release.bat* for Windows, *build_linux_release.bat* for Linux and Apple's OSX [not tested]) that will generate executable files) for Windows (outputs to directory *Windows_x64*), binary files for Linux (outputs to *Linux_x64*) and Mac together with all of the libraries needed into respective directory. For Mac, we included both *x86* and *arm* (outputs to OSX_x64 and OSX_arm respectively) architecture versions since at the time of writing this thesis both architectures are supported on Mac's OS. These directories will be available as a RAR file for download as attachment with this paper.

It is also possible to publish the programs for other architectures and systems (for more, see *dotnet publish* command documentation[3]), but we included these scripts for easier building for users on the currently most used OSs.

In the resulting directory (name depends on the target OS) we will find many files with extension *.dll* that represent all the libraries that the executables need to properly run and the executables themselves.

For **Windows** we can find executables *CartesianGP.exe* and *CombinedTreeBasedGP.exe*. For Linux and OSX the generated executables are generated *without an extension*, i.e. *CartesianGP* and *CombinedTreeBasedGP*.

Python

In this part, we will cover creating virtual environment, its activation and installation of modules from *requirements.txt* file.

First, we need to navigate to *python_scripts* directory using command line or terminal.

Figure 4.2: Example of path to *MastersProject\python_scripts*

Then we create virtual environment using module *virtualenv*. We download the module using command

pip install virtualenv

and now we are ready to create our virtual environment.

Type the following commands into the command line/terminal:

[3]https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-publish

1. *virtualenv venv*
2. activate environment
    - *venv\Scripts\activate.bat* (Windows specific)
    - *source venv/bin/activate* (Linux specific)
3. *pip install -r requirements.txt*

The first command (1) creates and stores the virtual environment into *venv* directory together with scripts used to activate and deactivate the environment. This way we can use *pip* as if we installed module onto the system.

The second command (2) activates the environment on Windows OS. The other command (2) show the way of activating the environment in Linux OS. Apple's OSX is absent since we were unable to test this during development. The command line/terminal should now start with "(venv)" as can be seen on figure below.

Figure 4.3: Example of command line after activating the environment

The last command (3) installs modules into the environment as described in *requirements.txt* from PyPi index[4], the official public repository of modules.

4.2 How to run

In this chapter we provide user guide on how to run GP programs including their command-line flags that change functionality of the program without the need to recompile it.

4.2.1 Cartesian GP

Command-line arguments

All command-line arguments with their description and default values can be seen when running the executable with *--help* flag. These arguments provide options of changing the algorithm's functionality without the need of recompiling it or understanding the code. This program implements Cartesian encoding described in 3.2.

- *--multi-threaded* is a boolean flag to enable running algorithm using multiple threads. Default is *false*.
- *--json* is used for loading arguments from JSON file. If used in combination with other arguments, arguments from command-line are prioritized.

[4]https://www.pypi.org

- *--train-csv* specifies path to CSV file used for training the model. It is **required** and expects the file to have headers.
- *--test-csv* specifies path to CSV file used for testing the model. It expects the file to have headers. If not provided, *--train-csv* is used to calculate final accuracy.
- *--csv-inputs-amount* specifies number of inputs. Due to loading dataset from single CSV file, we need to be able to distinguish between inputs and outputs per each line. This argument is **required**.
- *--csv-delimiter* is used to parse data from CSV files. CSV file does not have to separate values by comma, so we provided option to change the delimiter/separator. Default value is comma ",".
- *--min-threads* is used to specify minimum a number of threads used by .NET's ThreadPool class. It expects positive integer. Default is two.
- *--max-threads* is used to specify maximum a number of threads used by .NET's ThreadPool class. It expects positive integer. Default is four. This is also used in in PLINQ as *Degree of parallelism*.
- *--population-size* describes the size of population of a generation. It is expected to be *even and greater than 10*. Default value is 50.
- *--max-generations* describes amount of generations to evolve. Excepts positive integer greater or equal to hundred.
- *--repeat-amount* is used to run the GA algorithm multiple times. The main usecase is to test general performance of the algorithm since we are using multi threading, we are not able to use seed effectively.
- *--crossover-probability* is used to describe a probability with which *any* crossover occurs. Expects real number between zero and one. Default is 0.4.
- *--layer-sizes* describes starting sizes of each layer in Cartesian chromosome *excluding* input and output layer. Expects a list of integers separated by space. Default is "50 50".
- *--population-combination* is used to choose a strategy of combining previous and next generation. We have options "take-new" (take only the generation), "elitism" (take the best from previous generation and $(population - 1)$ from new generation) and "combine" (take the best individuals from both generations combined). Default is "take-new".
- *--change-node-mutation-probability* is used as probability of using ChangeNode mutation. Expects positive real number. Default is *zero*.
- *--change-parents-mutation-probability* is used as probability of using ChangeParents mutation. Expects positive real number. Default is *zero*.

- *--add-node-to-layer-mutation-probability* is used as probability of using AddNoteToLayer mutation. Expects positive real number. Default is *zero*.
- *--add-layer-mutation-probability* is used as probability of using AddLayer mutation. Expects positive real number. Default is *zero*.
- *--remove-node-to-layer-mutation-probability* is used as probability of using RemoveNoteToLayer mutation. Expects positive real number. Default is *zero*.
- *--remove-layer-mutation-probability* is used as probability of using RemoveLayer mutation. Expects positive real number. Default is *zero*.
- *--percentage-to-change* describes how much of an individual should change if the mutation takes place. Expects real positive number in interval $(0, 1)$. Default is 0.2.
- *--terminal-nodes-probability* describes probability of choosing a terminal node instead of non-terminal node. Expects real number in interval $(0, 1)$. Default is 0.2.
- *--value-node-weight* is used to describe weight of choosing Value node. Expects real positive value. Default is 0.2.
- *--sum-node-weight* is used to describe weight of choosing Sum node $(x + y)$. Expects real positive value. Default is 0.2.
- *--prod-node-weight* is used to describe weight of choosing Product node $(x * y)$. Expects real positive value. Default is 0.2.
- *--sin-node-weight* is used to describe weight of choosing Sin node $(sin(x))$. Expects real positive value. Default is 0.2.
- *--pow-node-weight* is used to describe weight of choosing Power node (x^y). Expects real positive value. Default is 0.2.
- *--unary-minus-node-weight* is used to describe weight of choosing UnaryMinus node $(-x)$. Expects real positive value. Default is 0.2.
- *--sig-node-weight* is used to describe weight of choosing Sigmoid node $(\sigma(x))$. Expects real positive value. Default is 0.2.
- *--relu-node-weight* is used to describe weight of choosing ReLU node $(ReLU(x))$. Expects real positive value. Default is 0.2.
- *--cond-node-weight* is used to describe weight of choosing Condition node $(if\ x > 0\ then\ y\ else\ z)$. Expects real positive value. Default is 0.2.

```
Windows_x64\CartesianGP.exe --train-csv prepared_train_iris_sklearn.csv --csv-delimiter , --csv-inputs-amount 4 --test-csv prepared_test_iris_sklearn.csv --population-size 50 --max-generations 1000 --percentage-to-change 0.2 --crossover-probability 0.2 --terminal-nodes-probability 0.1 --min-threads 2 --max-threads 4 --layer-sizes 20 50 100 --change-node-mutation-probability 0.4 --change-parents-mutation-probability 0.3 --population-combination elitism --add-node-to-layer-mutation-probability 0.1
```

Figure 4.4: Example of script running CartesianGP with arguments

Outputs

Each execution of CartesianGP program generates a new directory in format "cartesian-[year]-[month]-[day]_[hour]-[minute]-[second]". It also stores JSON file containing values of command-line arguments for future reference (config.json) and a short text file (info.txt) containing name of CSV file used for training with amount of output classes. If the algorithm runs multiple times (*--repeats-amount* argument is bigger than 1) it also creates subfolders in format "run_[run index]". Each of these directories contains files *result_formulas.txt* and *run.csv*. File *result_formulas.txt* contains final formulas for each of the output classes and accuracy separated by an empty line and *run.csv* contains statistics from the run of the algorithm needed for its analysis.

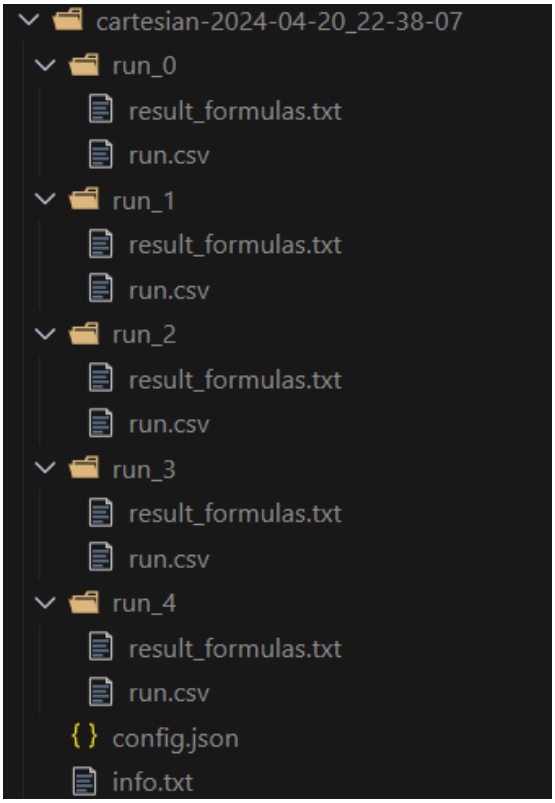

Figure 4.5: Example of output directory structure

4.2.2 Combined TreeBased GP

Command-line arguments

All command-line arguments with their description and default values can be seen when running the executable with *--help* flag. All arguments (except --multi-threaded boolean flag) expect value after the flag. This program implements tree-based encoding described in 3.3 in an ensemble (for each output class we create one tree).

- *--multi-threaded* is a boolean flag to enable running algorithm using multiple threads. Default is *false*.
- *--json* is used for loading arguments from JSON file. If used in combination with other arguments, arguments from command-line are prioritized.
- *--train-csv* specifies path to CSV file used for training the model. It is **required** and expects the file to have headers.
- *--test-csv* specifies path to CSV file used for testing the model. It expects the file to have headers. If not provided, *--train-csv* is used to calculate final accuracy.
- *--csv-inputs-amount* specifies number of inputs. Due to loading dataset from single CSV file, we need to be able to distinguish between inputs and outputs per each line. This argument is **required**.

- *--csv-delimiter* is used to parse data from CSV files. CSV file does not have to separate values by comma, so we provided option to change the delimiter/separator. Default value is comma ",".
- *--min-threads* is used to specify minimum number of threads used by .NET's ThreadPool class. It expects positive integer. Default is two.
- *--max-threads* is used to specify maximum number of threads used by .NET's ThreadPool class. It expects positive integer. Default is four. This is also used in in PLINQ as *Degree of parallelism*.
- *--population-size* describes the size of population of a generation. It is expected to be *even and greater than 10*. Default value is 50.
- *--max-generations* describes amount of generations to evolve. Excepts positive integer greater or equal to hundred.
- *--repeat-amount* is used to run the GA algorithm multiple times. The main usecase is to test general performance of the algorithm since we are using multi threading, we are not able to use seed effectively.
- *--crossover-probability* is used to describe a probability with which *any* crossover occurs. Expects real number between zero and one. Default is 0.4.
- *--population-combination* is used to choose a strategy of combining previous and next generation. We have options "take-new" (take only the generation), "elitism" (take the best from previous generation and $(population - 1)$ from new generation) and "combine" (take the best individuals from both generations combined). Default is "take-new".
- *--depth* describes default depth of tree when creating it. Expects positive integer. Default is three.
- *--change-node-mutation-probability* is used as probability of using ChangeNode mutation. Expects positive real number. Default is zero.
- *--shuffle-children-mutation-probability* is used as probability of using ShuffleChildren mutation. Expects positive real number. Default is zero.
- *--percentage-to-change* describes how much of an individual should change if the mutation takes place. Expects real positive number in interval $(0, 1)$. Default is 0.2.
- *--terminal-nodes-probability* describes probability of choosing a terminal node instead of non-terminal node. Expects real number in interval $(0, 1)$. Default is 0.2.
- *--value-node-weight* is used to describe weight of choosing Value node. Expects real positive value. Default is 0.2.
- *--input-node-weight* is used to describe weight of choosing Input node. Expects real positive value. Default is 0.2.

- *--sum-node-weight* is used to describe weight of choosing Sum node $(x + y)$. Expects real positive value. Default is 0.2.
- *--prod-node-weight* is used to describe weight of choosing Product node $(x * y)$. Expects real positive value. Default is 0.2.
- *--sin-node-weight* is used to describe weight of choosing Sin node $(sin(x))$. Expects real positive value. Default is 0.2.
- *--pow-node-weight* is used to describe weight of choosing Power node (x^y). Expects real positive value. Default is 0.2.
- *--unary-minus-node-weight* is used to describe weight of choosing UnaryMinus node $(-x)$. Expects real positive value. Default is 0.2.
- *--sig-node-weight* is used to describe weight of choosing Sigmoid node $(\sigma(x))$. Expects real positive value. Default is 0.2.
- *--relu-node-weight* is used to describe weight of choosing ReLU node $(ReLU(x))$. Expects real positive value. Default is 0.2.
- *--cond-node-weight* is used to describe weight of choosing Condition node $(if\ x > 0\ then\ y\ else\ z)$. Expects real positive value. Default is 0.2.

```
1  Windows_x64\CombinedTreeBasedGP.exe --multi-threaded -
   -input-csv prepared_mnist_sklearn.csv --input-csv-inpu
   ts-amount 64 --population-size 50 --max-generations 10
   0 --depth 7 --percentage-to-change 0.1 --crossover-pro
   bability 0.2 --mutation-probability 0.4 --terminal-nod
   es-probability 0.1 --min-threads 6 --max-threads 6
```

Figure 4.6: Example of script running CombinedTreeBasedGP with arguments

Outputs

Each execution of CartesianGP program generates a new directory in format "combined-[year]-[month]-[day]_[hour]-[minute]-[second]". It also stores JSON file containing values of command-line arguments for future reference (config.json) and a short text file (info.txt) containing name of CSV file used for training with amount of output classes. If the algorithm runs multiple times (*--repeats-amount* argument is bigger than 1) it also creates subfolders in format "run_[run index]". Each of these directories contains files *result_formulas.txt* and *run.csv*. File *result_formulas.txt* contains final formulas for each of the output classes and

accuracy separated by an empty line and *run.csv* contains statistics from the run of the algorithm needed for its analysis.

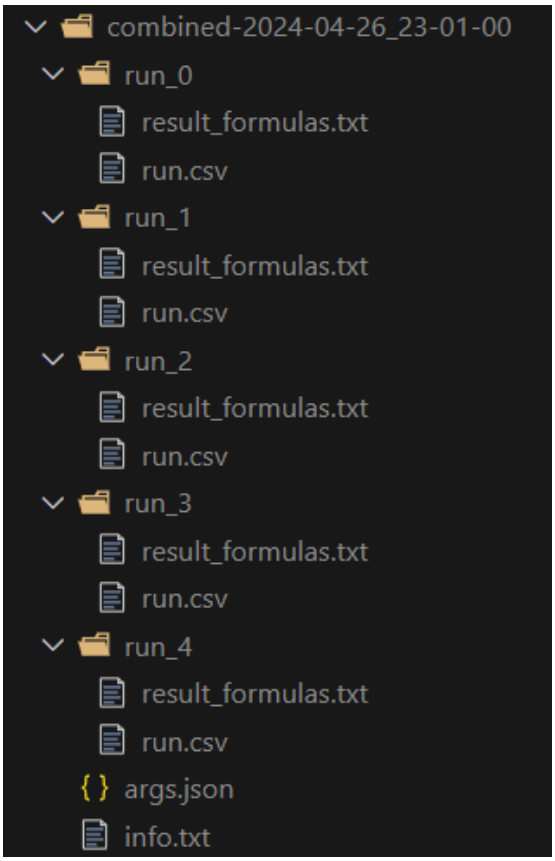

Figure 4.7: Example of output directory structure

4.2.3 Python scripts

Dataset creation

The following scripts are used to load a dataset from *scikit-learn* library and save it as CSV file in the current directory:

- *iris_prepare.py* creates file *iris_sklearn.csv*.
- *breast_cancer_prepare.py* creates file *breast_cancer_sklearn.csv*
- *wine_prepare.py* creates file *wine_sklearn.csv*
- *mnist_prepare.py* creates file *mnist_sklearn.csv*

prepare_input.py

This script is used to prepare dataset to be used by C# program. When running this script with command-line flag *--help* we can see explanation of all command-line arguments including their default values. This script also needs a positional argument of path to CSV files of input dataset.

- *-d* or *--delimiter* is used when loading CSV file since the separator can be different. Default is ",".
- *-s* or *--seed* is used to set seed splitting method for separating dataset into *train* and *test* datasets.
- *--index-col* can be specified for loading dataset from CSV file.
- *--one-hot* is a boolean flag that converts all categorical columns to one-hot encoded columns. This also expects last column to be categorical. **It is expected to be used when creating dataset/-s for C# programs**.
- *--include-index* signals if index column should be present in output CSV files.
- *--train-ratio* indicates how much of dataset should be used as dataset for training. This argument expects number in interval [0,1) Test ratio is complement to this number $(1 - train\_ratio)$.

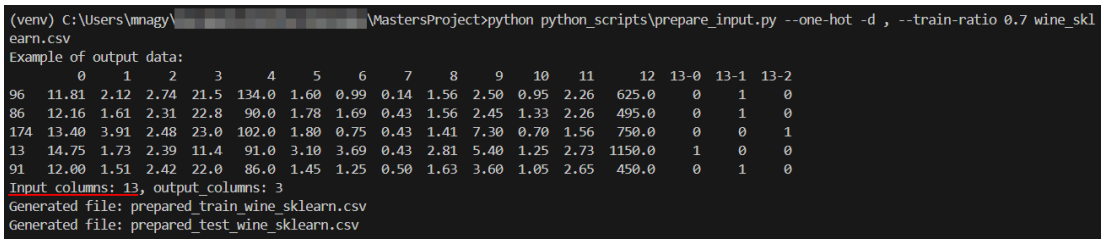

Figure 4.8: Example of running script prepare_input.py

We need the underlined information for running the C# programs (*--csv-inputs*)

show_multiple_runs.py

This script builds and shows graphs that describe how fitness, score and depth changed over time of running the GA. We can use the following command-line flags:

1. *-d* or *--directory* should contain a path to directory with all output files from a GA that has run.
2. *-s* or *--save* is a boolean flag where instead of showing them, the script will save the figures to the *directory specified in flag --directory* as files *fitness_progress.png*, *score_progress.png* and *depth_progress.png*.

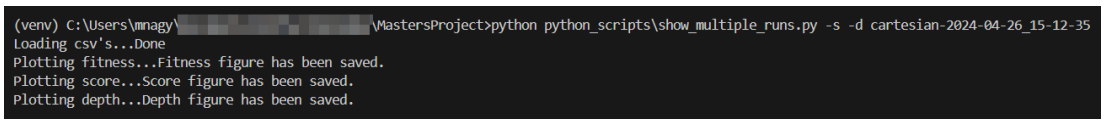

Figure 4.9: Example of running script show_multiple_runs.py

5. Experiments

We run both *cartesian* and *combined tree-based* GP on multiple classification datasets and evaluate their performance both overall and against each other. We also show graph of evolution of populations over runtime of a specified algorithm where the line is mean across multiple runs (*--repeat-amount*) and shade is *95% confidence interval* over the mentioned runs.

Regarding reproducibility, these programs were designed to function using multi-threading and so we chose to use .NET's *Random.Shared* global object that is secure to use in this kind of environment, but we loose the ability to exactly reproduce these experiments and so we run experiments multiple times (default is 5) in order to achieve statistical relevance.

We also omit graph of depth for CartesianGP in all experiments and mention it in the text. This decision was made since we achieved satisfactory results without using mutations modifying of the architecture of the chromosomes (AddNodeToLayer, AddLayer, RemoveNodeFromLayer, RemoveLayer).

5.1 Datasets used

We are using datasets included in Python *scikit-learn*[1] library in its submodule datasets. We are using all provided datasets *used for classification*: iris, breast cancer, wine, and MNIST.

We have divided these datasets into train and test datasets using provided script *prepare_input.py* with train-ratio being 70%, i.e. train dataset contains 70% of the original *stratified* dataset. We show multiple graphs where on each the X axis represents number of generation and therefore evolution of solutions through time. First, we show the evolution of fitness, then depth and at the end of each experiment's section we show the best evolved results.

5.1.1 Iris dataset

This dataset is considered very small and it is used primarily as a check that the ML method developed actually works at all. It consists of four input variables that are real positive numbers and we classify into three classes. It has 150 total samples, 50 for each class.

We are using population size of 50 individuals and maximum generations of 1000. For more details about configuration, CartesianGP was run using Windows command-line script *run_cartesian_iris.bat* and for CombinedTreeBasedGP we used *run_treebased_combined_iris.bat*, both provided in the attachment to this thesis.

[1] https://scikit-learn.org/stable/

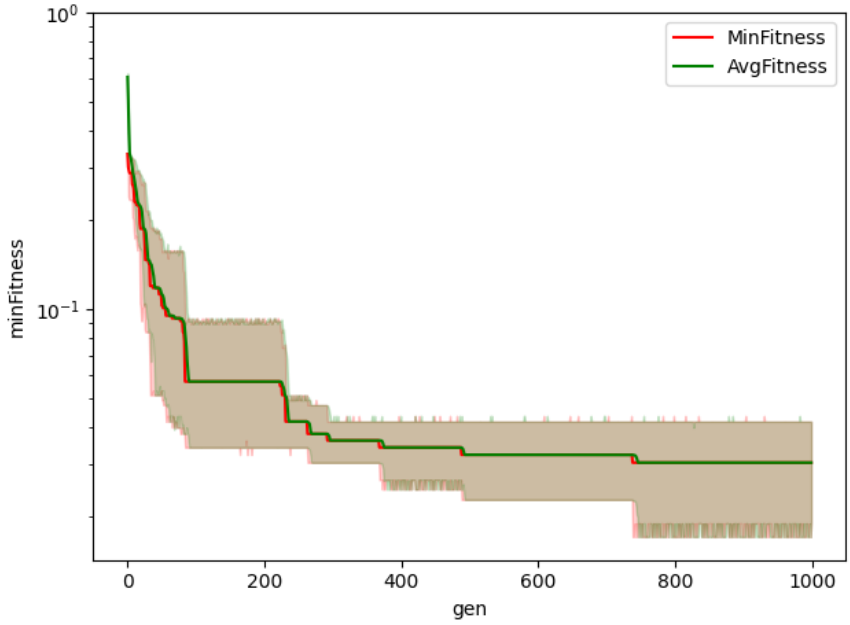

Figure 5.1: Fitness progress in CartesianGP on Iris dataset.
Axis X represents number of generations, axis Y is logarithmically scaled fitness.

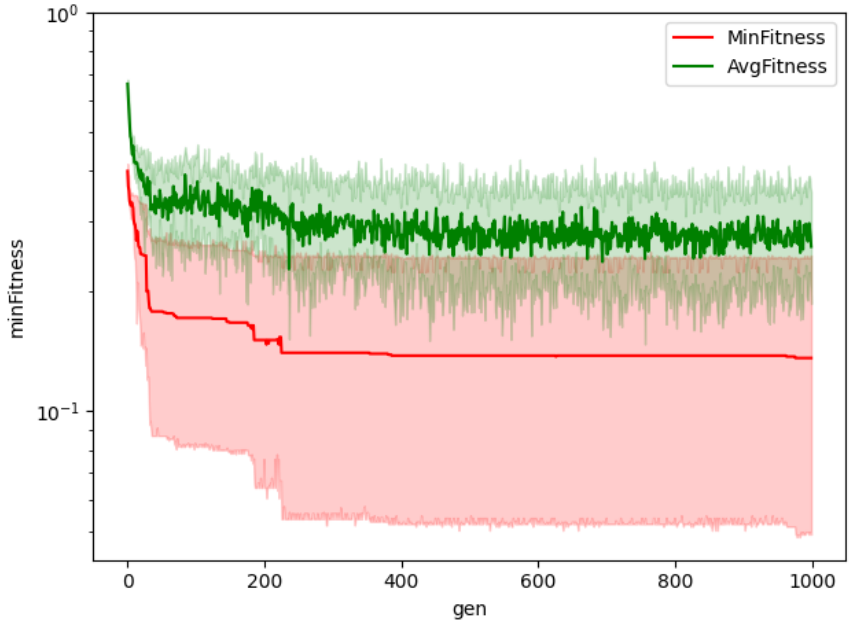

Figure 5.2: Fitness progress in CombinedTreeBasedGP on Iris dataset with log scaling.
Axis X represents number of generations, axis Y is logarithmically scaled fitness.

We can clearly see that CartesianGP (figure 5.1) is improving during the algorithm and converges in about 800 generations. CombinedTreeBasedGP (figure 5.2) in about 200 generations and does not significantly improve beyond this

value. Since the fitness is combined using multiple factors (accuracy, depth of the models) it is not clear what algorithm results in better results. In the next part, we analyse depth of evolved individuals during run of the algorithm.

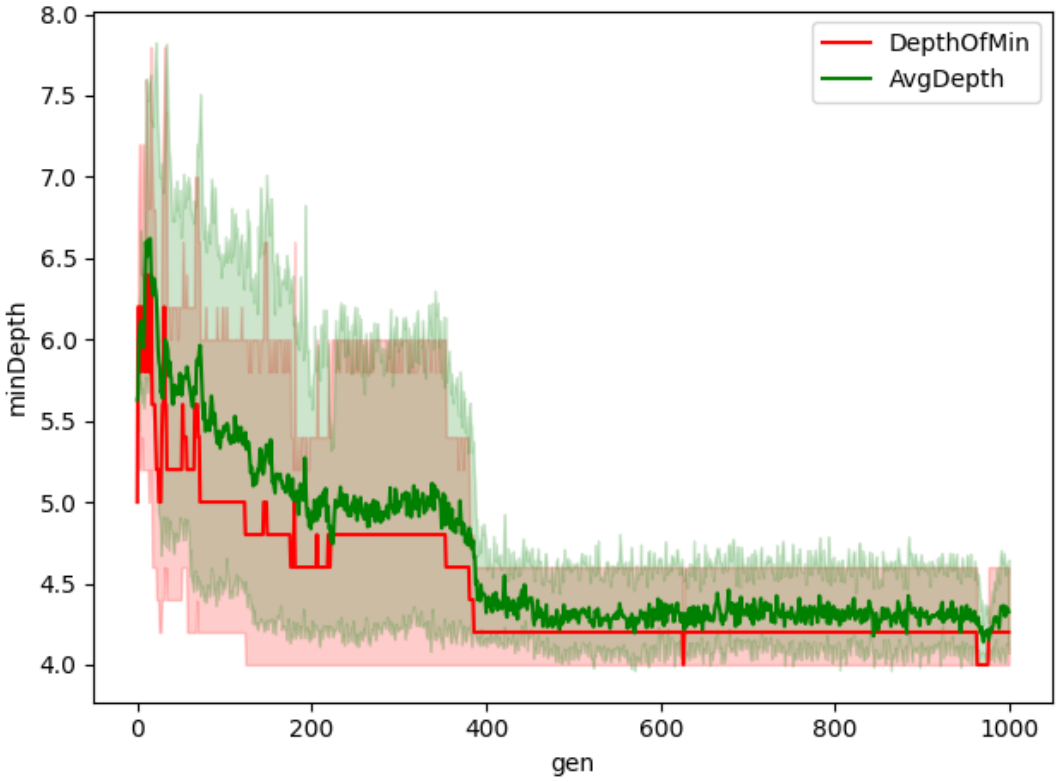

Figure 5.3: Depth evolution for CombinedTreeBasedGP on Iris dataset. Axis X represents number of generations, axis Y is depth of the deepest tree in an individual.

Since we decided to not include mutations that modify architecture of CartesianGP chromosomes, depth of CartesianGP is constant throughout the algorithm, *five* to be exact. This *includes* input and output layers. We can see that CombinedTreeBasedGP's depth (figure 5.3) lowered over time and settled between four and five while maintaining similar fitness. Now, we take a look at the results themselves where each formula represents one output class (the class with the biggest value is chosen as prediction), and the accuracy is computed on test dataset.

```
1  sin(ReLU(x_2))
2  sigmoid(sin((ReLU(x_2)+x_2)))
3  (((-(x_1))+(x_3*x_3))*ReLU(ReLU((x_0+x_2))))
4
5  Accuracy score: 99.04761904761905 %
```

Figure 5.4: The best results for CartesianGP on Iris dataset.

```
1  sin(x_2)
2  ((x_1)^(sin((4)+(x_2))))^(sin(2))
3  x_3
4
5  Accuracy score: 97.14285714285714 %
```

Figure 5.5: The best results for CombinedTreeBasedGP on Iris dataset.

CartesianGP (figure 5.4) achieves better results then CombinedTreeBasedGP (figure 5.5) achieved on test dataset. CombinedTreeBasedGP evolved clearer and shorter formulas compared to CartesianGP. The CombinedTreeBasedGP does not steadily improve over multiple runs so we can see from fitness graph it has a big variance between runs. The end result is: CombinedTreeBasedGP *can* achieve better result, but CartesianGP is *more stable* algorithm.

5.1.2 Breast cancer dataset

This dataset is still considered small and it is used as a next step in increasing difficulty for our ML model. It consists of 30 input variables that are real positive numbers and we classify into two classes. It has 570 total samples, 212 for *malignant* class and 357 for *benign* class.

We are using population size of 50 individuals and maximum generations of 1000. For more details about configuration, CartesianGP was run using Windows command-line script *run_cartesian_breast_cancer.bat* and for CombinedTreeBasedGP we used *run_treebased_combined_breast_cancer.bat*, both provided in the attachment to this thesis.

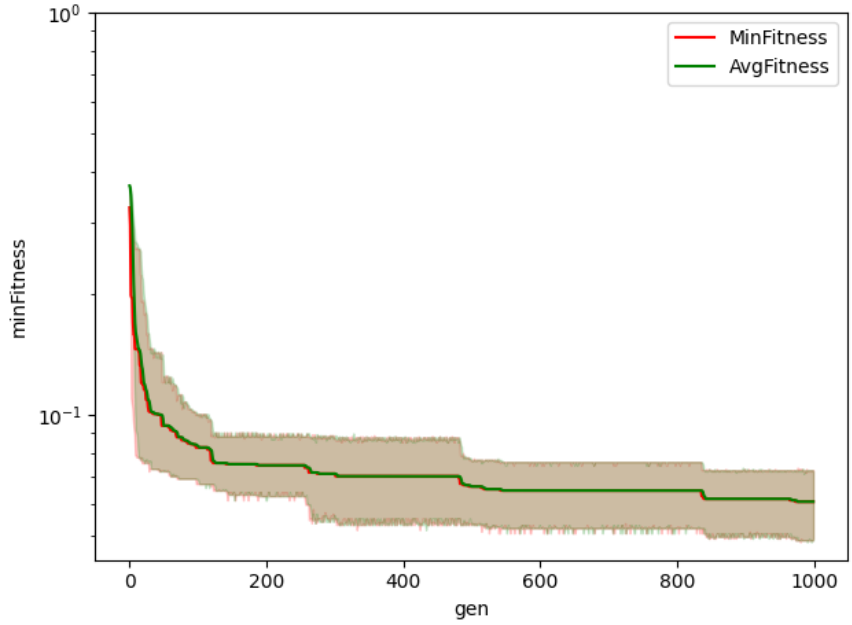

Figure 5.6: Fitness progress in CartesianGP on Breast cancer dataset. Axis X represents number of generations, axis Y is logarithmically scaled fitness.

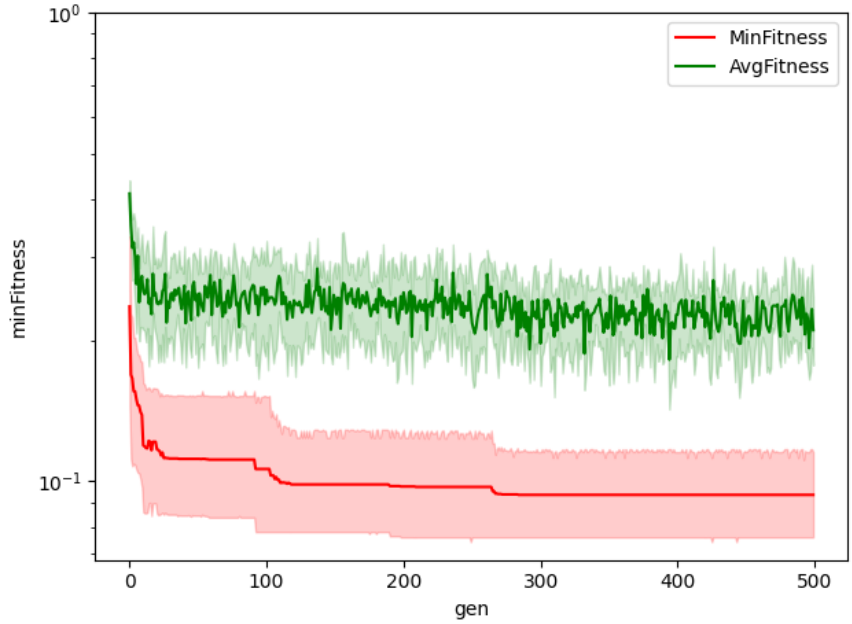

Figure 5.7: Fitness progress in CombinedTreeBasedGP on Breast cancer dataset. Axis X represents number of generations, axis Y is logarithmically scaled fitness.

We again see that CartesianGP (figure 5.6) is more consistent in achieving better fitness values with a smaller variance between the performance of chromosomes from different runs when compared to CombinedTreeBasedGP (figure 5.7). Next, we analyse depth evolution of the same runs.

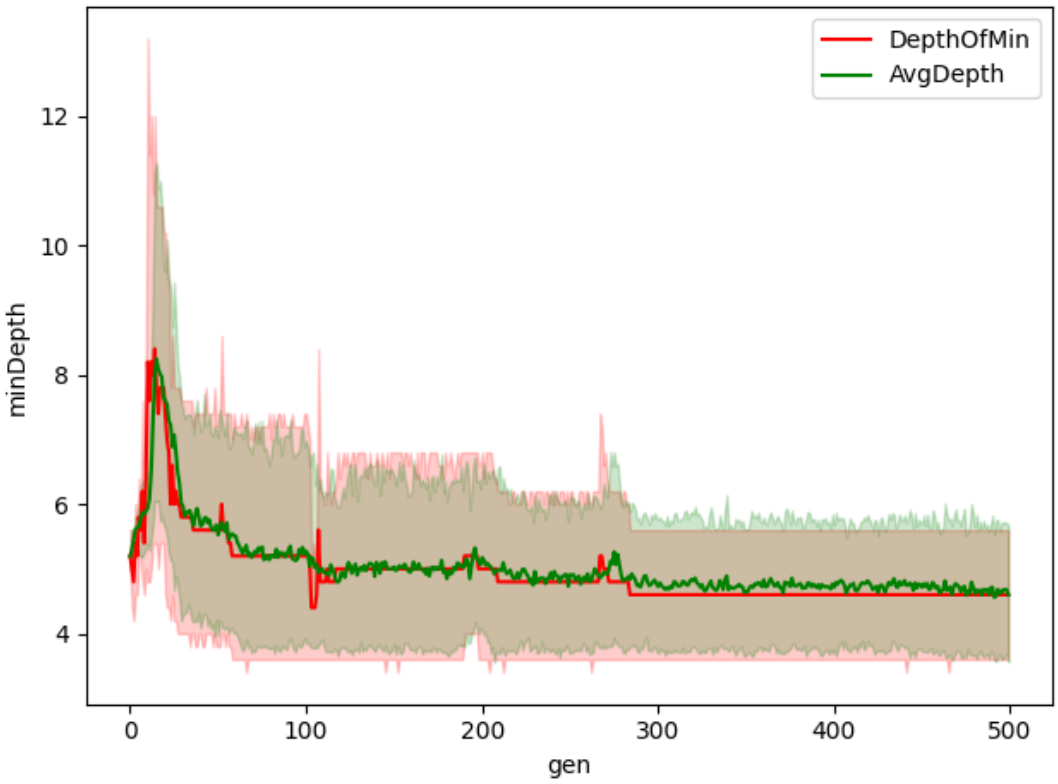

Figure 5.8: Depth evolution for CombinedTreeBasedGP on Breast cancer dataset. Axis X represents number of generations, axis Y is depth of the deepest tree in an individual.

For CartesianGP we use depth of nine *including* input and output layers. CombinedTreeBasedGP (figure 5.8) on average tends to stay around the depth of five, but with some variance between runs of the algorithm. Interesting is also the initial spike in depth values across runs and almost immediately return to the original depth of around five. In the next part, we take a look at results themselves with accuracies on test datasets.

```
1  ((((x_27+x_26)+x_26)*((x_21+x_1)*x_7))*ReLU([if
   sigmoid(x_15) > 0) then (sin(x_7)) else (x_8)]))

2  (sin(sigmoid(x_20))*sin(x_5))

3

4  Accuracy score: 95.7286432160804 %
```

Figure 5.9: The best *result_formulas.txt* for CartesianGP on Breast cancer dataset.

```
1  sigmoid((1)+(x_29))
2  ((x_9)^(x_10))^(x_26)
3
4  Accuracy score: 93.21608040201005 %
```

Figure 5.10: The best *result_formulas.txt* for CombinedTreeBasedGP on Breast cancer dataset.

We see that CartesianGP (figure 5.9) achieves higher accuracy, but only by about 2% on test dataset and CombinedTreeBasedGP provides a simpler formula. CombinedTreeBasedGP also has bigger variance between runs *and* between average fitness and minimal fitness in each generation (figures 5.6 and 5.7).

5.1.3 Wine dataset

This dataset is used as a next step in increasing difficulty for our ML model. It consists of 13 input variables that are real positive numbers and we classify into two classes. It has 178 total samples, 59 for the first class, 71 for the second class and 48 for the third class.

We are using population size of 50 individuals and maximum generations of 1000. For more details about configuration, CartesianGP was run using Windows command-line script *run_cartesian_wine.bat* and for CombinedTreeBasedGP we used *run_treebased_combined_wine.bat*, both provided in the attachment to this thesis.

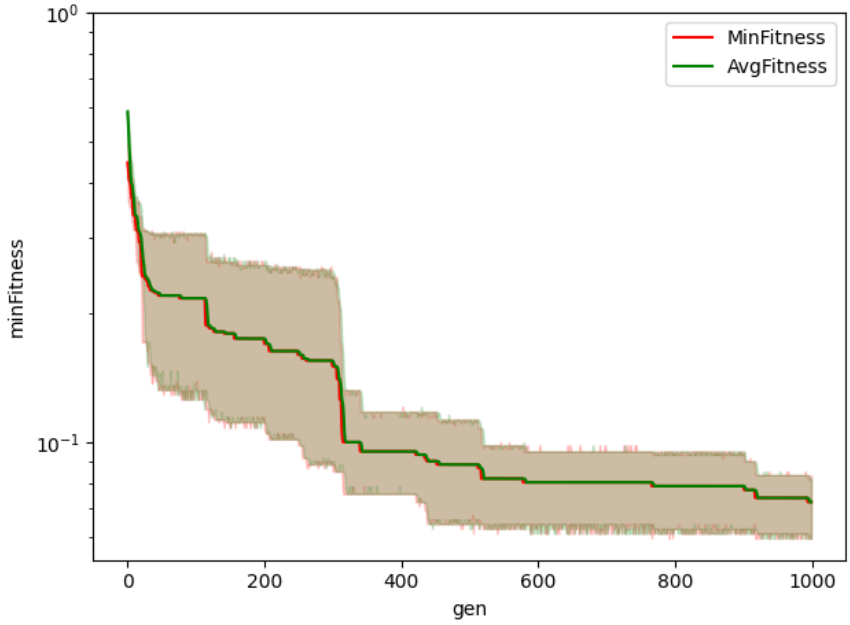

Figure 5.11: Fitness progress in CartesianGP on Wine dataset.
Axis X represents number of generations, axis Y is logarithmically scaled fitness.

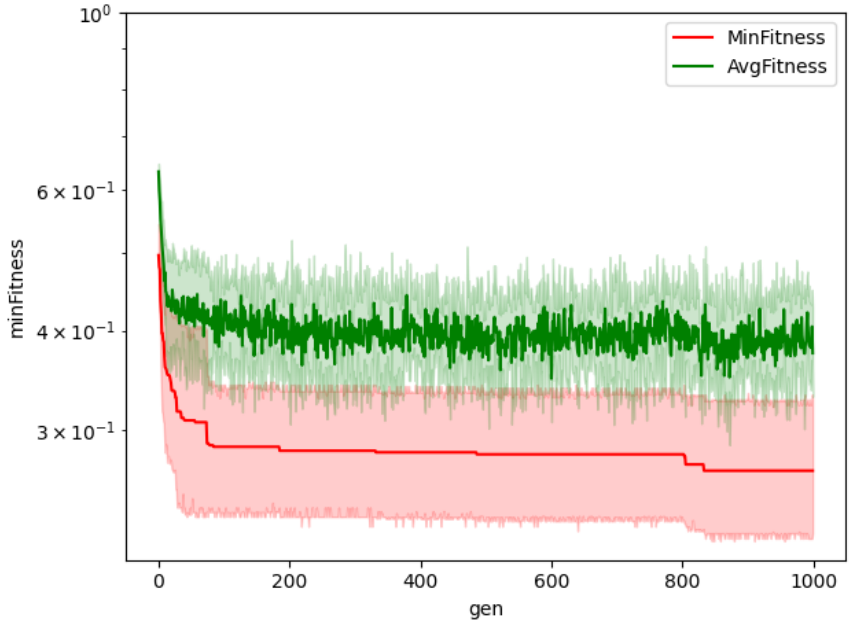

Figure 5.12: Fitness progress in CombinedTreeBasedGP on Wine dataset.
Axis X represents number of generations, axis Y is logarithmically scaled fitness.

Again, we see that CartesianGP (figure 5.11) improves over the whole time of the evolution whereas CombinedTreeBasedGP (figure 5.12) experiences initial jump in fitness and then does not experience significant improvement during the

whole run of the algorithm. In the next part, we analyse the evolution of the depth of the chromosomes over the run of the algorithms.

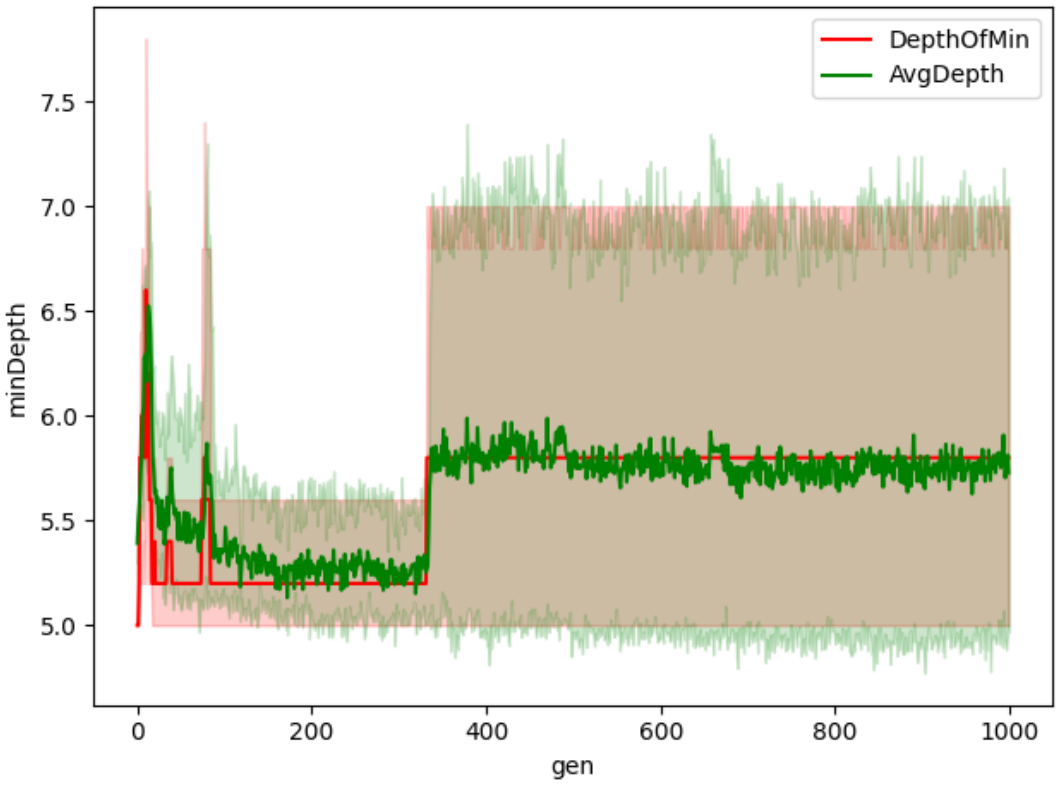

Figure 5.13: Depth evolution for CombinedTreeBasedGP on Wine dataset. Axis X represents number of generations, axis Y is depth of the deepest tree in an individual.

CartesinGP has a constant depth of six. We see that CombinedTreeBasedGP's average depth (figure 5.13) stays around five initially, but in around generation 300 the variance over runs widens significantly. In the next part, we take a look at the best result out of the five runs of the algorithm.

```
1  (sin(ReLU(x_0))*(-((-(x_8)))))
2  sin(x_6)
3  ((-((x_11*x_5)))+ReLU(x_9))
4
5  Accuracy score: 95.16129032258065 %
```

Figure 5.14: The best results for CartesianGP on Wine dataset.

```
1  (-(x_10))+(x_6)
2  x_10
3  -(-((sigmoid(x_11))^(x_10)))
4
5  Accuracy score: 81.45161290322581 %
```

Figure 5.15: The best results for CombinedTreeBasedGP on Wine dataset.

We see that on Wine dataset, CombinedTreeBasedGP generates result (figure 5.15) with lower accuracy on test dataset when compared to CartesianGP's result (figure 5.14). In this instance, both sets of generated formulas are easy to read.

5.1.4 MNIST dataset

This dataset is the hardest one we used for testing. It is a basic image recognition dataset for 8x8 greyscale picture where each pixel is represented by a value from 0 up 16 (included). It consists of 64 integer inputs from interval $[0, 16]$ and it has 10 output classes. It contains 1800 total samples, 180 for each class.

CartesianGP was run using Windows command-line script *run_cartesian_mnist.bat* and for CombinedTreeBasedGP we used *run_treebased_combined_mnist.bat*, both provided in the attachment to this thesis.

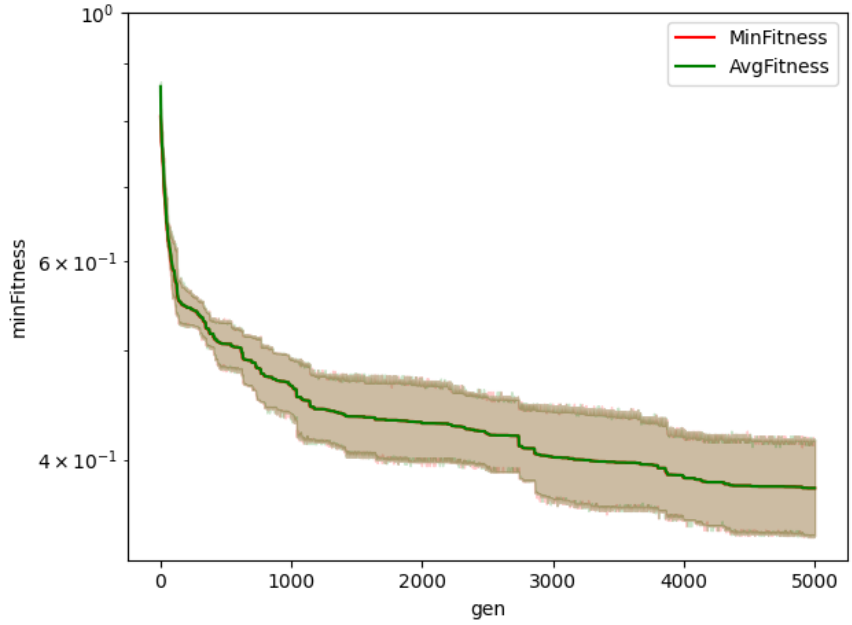

Figure 5.16: Fitness progress in CartesianGP on MNIST dataset. Axis X represents number of generations, axis Y is logarithmically scaled fitness.

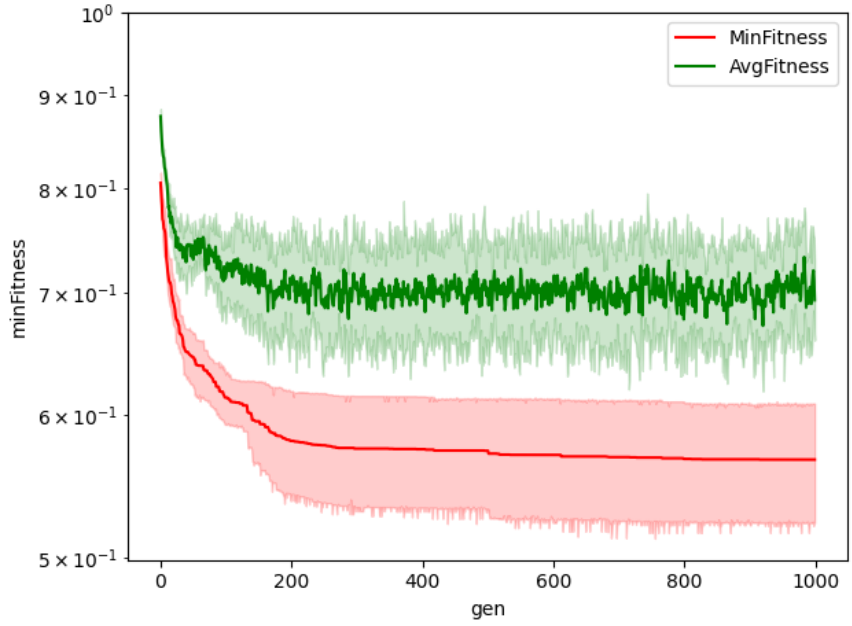

Figure 5.17: Fitness progress in CombinedTreeBasedGP on MNIST dataset. Axis X represents number of generations, axis Y is logarithmically scaled fitness.

We can see that CartesianGP (figure 5.16) steadily improves and we can see that its fitness regularly achieves values around 0.4. We again see that variance of the best chromosomes across runs out of the population varies by about 0.1

fitness which is big. In the next part, we analyse the evolution of the depth of the chromosomes over the run of the algorithms.

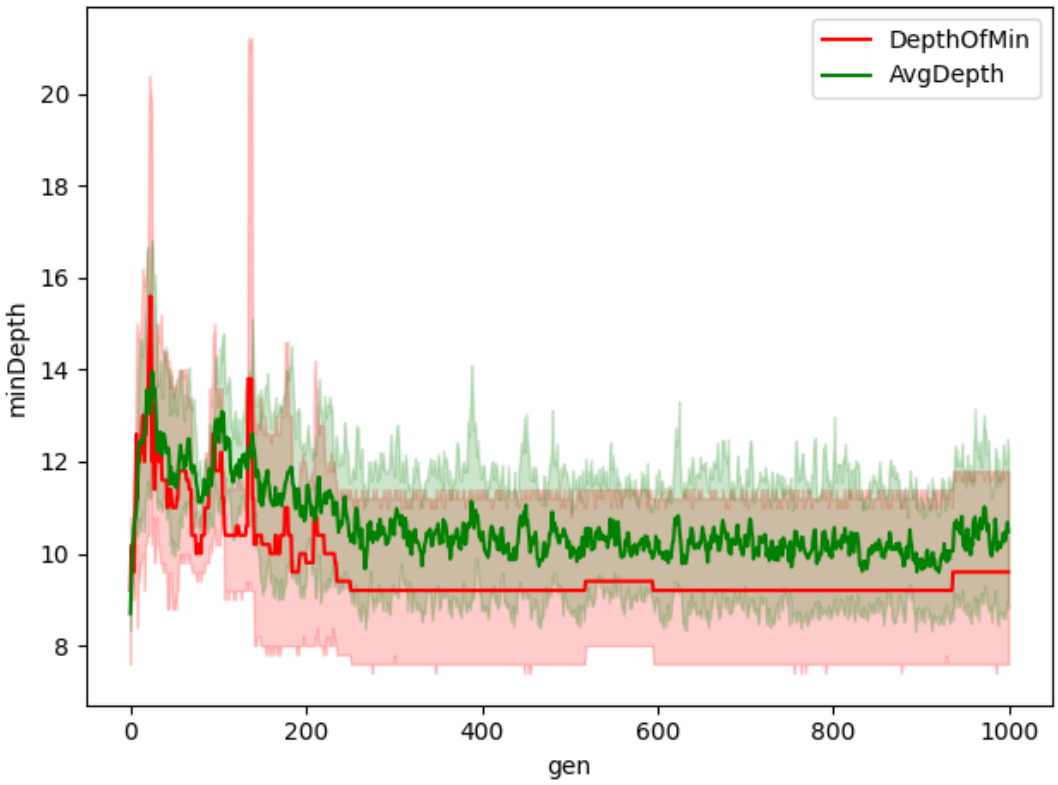

Figure 5.18: Depth evolution for CombinedTreeBasedGP on MNIST dataset. Axis X represents number of generations, axis Y is depth of the deepest tree in an individual.

CartesianGP was set to constant depth of nine including input and output layers. CombinedTreeBasedGP's depth (figure 5.18) again varied a lot, but we see that in the beginning chromosomes got deeper to achieve better results and afterwards came back to average depth of 10. In the next part, we take a look at the best produced results from each GP algorithm.

```
(((x_51*x_30)+(-(x_27)))*(x_13*x_33))

2

(x_62*x_3)

(((-(x_46))*x_26)+ReLU((x_54*x_45)))

((sigmoid((-(x_46)))+1)+ReLU(x_38))

[if (-(x_54)) > 0) then (sin((x_2*sigmoid(x_2)))) else (ReLU(x_2))]

((-([if x_18 > 0) then ([if x_56 > 0) then (x_12) else (x_34)]) else ([if x_26 > 0) then (x_62) else (x_39)]))]))*(-(x_54)))

(((x_13*x_33)+x_30)*([if x_45 > 0) then (x_7) else (x_21)]*[if x_28 > 0) then (x_43) else (x_29)]))

(sigmoid((-(x_33)))*(((x_36*x_21)+ReLU(x_7))*sigmoid(sigmoid(x_2))))

(((sigmoid((-(x_44)))*((x_21*x_21)+(-(x_43))))+sigmoid(x_48))+sigmoid(x_42))


Accuracy score: 66.26889419252187 %
```

Figure 5.19: The best results for CartesianGP on MNIST dataset.

```
(((x_42)*(x_30))*(sin(3)))+(-(1))

(ReLU(x_19))+(-((ReLU(ReLU(5)))+(if ((ReLU(x_53))*(3) > 0) then [(sin(2))+(if (x_25 > 0) then [x_10] else [ReLU(x_53)])] else [((-(x_6))+(2))^(2)])))

sin(sigmoid(x_20))

sigmoid(x_37)

sin(x_40)

(ReLU(sin(((3)^(ReLU(3)))^(((0)+(5))^((x_61)+(x_61))))))^(ReLU(ReLU(x_20)))

((-(((x_58)+(x_21))*(sigmoid((5)+(5)))))+(sigmoid(x_61)))+((5)+(x_31))

ReLU(x_30)

x_31

sin(x_15)


Accuracy score: 51.86953062848051 %
```

Figure 5.20: The best results for CombinedTreeBasedGP on MNIST dataset.

We see that CartesianGP (figure 5.19) achieves significantly better accuracy on test dataset when compared to CombinedTreeBasedGP (figure 5.20). Since this is more complicated dataset, it is hard to argue which formulas are simpler or concise.

Conclusion

In the first chapter, we discussed theory behind Evolutionary algorithms, Genetic programming and classification including smaller parts necessary to be design for a specific problem the algorithm is trying to solve. In the second chapter, we introduced cartesian and tree-based encodings with functions specific for them: crossover, mutation, fitness function. We also described all node types we use and how they function. Chapters three to five present our original work and complete the goal of this thesis. In the third chapter, we presented our proposal of implementing both encodings with details regarding chromosome itself, fitness function, crossovers and mutations, and shared library responsible for handling the running of genetic algorithm. In the fourth chapter, we described how to compile, create a virtual environment and run implementation provided in attachments of this thesis including explanation of all command-line arguments. In the fifth chapter, we show results of our experiments on selected datasets and discuss the results.

In this thesis, we wanted to see if using GP as a machine learning classification model is a viable option, if we want more understandable results and/or formulas meaning this model is not a "black box" and after training we can analyse the models formulas. These solutions are represented by relatively simple formulas and are fast to compute once the training has ended. This has been achieved in chapter 5 and completed the goal of this thesis.

Possible future work in this area is testing these encodings for classification on larger datasets. Another possible improvement might come from adding more mutations, crossover functions and node types. and also if building on our implementation, take a look at efficiency or a possible data leak in implementation of CombinedTreeBasedGP (during experiments can take upto seven gigabytes of memory)

Bibliography

John Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, page 215. Morgan-Kaufmann, 1989. URL https://proceedings.neurips.cc/paper_files/paper/1989/file/0336dcbab05b9d5ad24f4333c7658a0e-Paper.pdf.

Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

John H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–73, 1992. ISSN 00368733, 19467087. URL http://www.jstor.org/stable/24939139.

John R. Koza. *Genetic programming: On the programming of computers by means of natural selection*. The MIT Press, 1992.

Microsoft. dotnet build, a. URL https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-build#description.

Microsoft. Introduction to plinq, b. URL https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq.

Microsoft. Threadpool class, c. URL https://learn.microsoft.com/en-us/dotnet/api/system.threading.threadpool?view=net-8.0.

Julian F. Miller. *Cartesian Genetic Programming*. Natural Computing Series. Springer, 2011. ISBN 978-3-642-17309-7. doi: 10.1007/978-3-642-17310-3. URL https://doi.org/10.1007/978-3-642-17310-3.

Mitchell Spryn. Cartesian genetic programming for image processing. URL https://www.mitchellspryn.com/2021/01/06/Cartesian-Genetic-Programming-For-Image-Processing.html.

List of Figures

A. Attachments

A.1 First Attachment

Electronic attachments to this thesis include the software implementation, data and results of the experiments from chapter 5. They are also available at GitHub: `https://github.com/marnagy/MastersProject`.