A Novel Multigene Genetic Programming System

Cole Corbett
Brock University
St. Catharines, Ontario, Canada
cc21gg@brocku.ca 7246325

Cameron Carvalho
Brock University
St. Catharines, Ontario, Canada
cc21lz@brocku.ca 7240450

Quinn Guernsey
Brock University
St. Catharines, Ontario, Canada
qg20wy@brocku.ca 7214075

***Abstract*—Darwinian-inspired algorithms have been proven to be effective search paradigms through years of implementation. These search paradigms have the ability to search a wide space while progressing towards local optima. Genetic Programming (GP) is one such paradigm. This paper discusses a novel system adapting the original GP paradigm, utilizing a different representation of individuals, this paradigm is called Multigene Genetic Programming (MGGP). Instead of representing individuals as a tree, this system represents individuals as a collection of weighted trees. This paper explores the implementation of the novel system with its required functionality and then explains how this system is used. The paper then goes on to perform two comparative experiments, determining the efficacy of this novel system in multiple key metrics. This paper concludes with some useful analyses and starting points for future research on MGGP.**

I. Introduction

MGGP is a modified GP framework. In regular GP systems, individuals are represented as a single tree. In MGGP, individuals are instead represented by a collection of genes with associated weights [8]. This report seeks to implement and test a MGGP system, in the Python programming language. As it stands there currently exists only one commonplace system for implementing MGGP; GPTIPS [8] which is a library for MATLAB; a subscription software. This report describes the implementation of a novel Python MGGP system, along with a user guide for setup. This report aims to provide a more accessible solution to MGGP. This report then attempts to test for any improvements in the MGGP system over a regular GP system Our 2 experimental setups seek to compare results using two unique problems. The first experiment is a symbolic regression which aims to evolve a function that closely resembles a given target function. A GP system is well suited to problem like this, so intrigue stems from how the quality of the solution may differ between the two systems unique approach. For the second experiment, using a classification dataset from UCI Machine Learning Repository [6], a dataset of several parametric measurements of grains of rice with two possible classifications of rice species were used to compare the regular GP system with the MGGP system. In this case, our metrics of interest that will be compared include minimizing incorrect classification of rice species, and the number of generations to evolve a good solution.

II. MGGP System Implementation

This MGGP system was implemented in Python 3.9, and works in Python versions 3.9 through 3.12. This system is built using numpy [9] and scikit-learn is recommended for gene weighting [7]. This system differs from other GP systems in their representation of an individual. Within this system, an individual is represented as a series of genes. Each gene is a unique tree representation of a classic GP program as described by Koza [3]. To achieve a final output, these genes are all evaluated individually on the terminal set, and a regression model attempts to combine the individual genes in such a way that they are fit to the target. This results in a final structure, where the intercept can be added to each tree multiplied by its respective weight to obtain the predicted value. This is described as follows:

$$y = \text{intercept} + w_1 \times \text{Tree}_1 + w_2 \times \text{Tree}_2 + \ldots + w_n \times \text{Tree}_n$$

This MGGP system is built using multiple essential parameters, classes, and functions, which are described as follows:

A. Parameter Description

seed
Provides a deterministic sequence for the system. Supplied as an integer.
pop_size
The number of individuals within each generation. Supplied as an integer.
num_genes
The maximum amount of genes an individual can possess. Supplied as an integer.
terminals
Names of variables that are replaced during evaluation. For example, x is a terminal in symbolic regression. Supplied as a list of strings.
minInitDepth
The minimum depth of each tree in the initial population. Supplied as an integer.
maxInitDepth
The maximum depth of each tree in the initial population. Supplied as an integer.
max_global_depth
Maximum global depth for trees within the program. Supplied as an integer.
max_crossover_growth
The maximum amount of crossover growth allowed. Supplied as an integer.
max_mutation_growth
The maximum amount of mutation growth allowed. Supplied

as an integer.
mutation_rate
The percentage chance of mutation. Supplied as a float.
crossover_rate
The percentage chance of crossover. Supplied as a float.
num_generations
The number of generations to iterate over. Supplied as an integer.
elitism_size
The number of elite individuals that carry over from each generation.
fitnessType
Allows dynamic minimization or maximization of the fitness by supplying "Minimize" or "Maximize".
ops
The operators or functions included within the tree structure. Supplied as a dictionary with the names and functions.
arity
The number of parameters required to utilize an operator. Supplied as a dictionary with the names and arity.

B. Node

As the foundational building blocks of this MGGP system, the Nodes serve as containers that hold an operator, terminal, or ephemeral value. Nodes can be evaluated to return the numerical value and parsed to return a string of the element. Working within a node hierarchy like a tree structure, these structures can be recursively evaluated to obtain a prediction.

C. Tournament Selection

Tournament selection chooses k random individuals within the population and returns the individual with the best fitness evaluation. These individuals chosen become the parents of the next generation's population.

D. One-Point Crossover

This MGGP system supplies one default crossover operator – one-point crossover. First, the crossover probability will be checked. If the resultant float is above the crossover probability, crossover will not occur, and a direct copy is added to the offspring. Otherwise, this operator selects two random individuals from the population using the selection function. A random gene is then chosen from each individual. Within each gene, a crossover point is selected. The nodes below the points in each gene are then exchanged, and the new genes replace the old genes within the individual. One-Point Crossover ensures that both the maximum crossover growth and maximum depth are not exceeded.

E. Mutation

This MGGP system supplies an internal method of mutating a gene, which is called using one of two external mutation operators.

To mutate a gene, the program will randomly select a node from within that gene, and generate a new tree from that node, replacing whatever children were there originally while maintaining both the global depth limit, and the max mutation growth limit.

To mutate an individual, the user must specify which of two external mutation operators to use. Both of these operators will first check the mutation probability. If the generated float is above the mutation probability the individual won't be mutated, instead, it will be directly copied from the offspring to the new population. Otherwise, the first mutation operator will select each gene within an individual, and perform the gene mutation operator on each gene. The second mutation operator will select a random gene within an individual and perform the gene mutation operator only on that specific gene.

F. Generate Tree

Generate tree is an algorithm that generates a full tree. This algorithm works by creating a root node, and then continually adding child nodes according to the arity of the last generated node. This algorithm has 2 termination criteria. First, at the maximum depth, it will generate a leaf node. Second, once the tree surpasses the minimum depth, there is a 50% chance it will generate a leaf node. Each child node added is selected from either a list of non-leaf nodes (user-supplied operators) or leaf nodes (terminals/ephemerals) depending on its node type. This function is used for mutation and population initialization.

G. Fitness Evaluation

The fitness evaluation is a user-defined function that must follow a schema as defined in the user guide. The fitness function performs four key steps. First, the individual trees should be evaluated on the data. Next, some way of fitting the individual to the output must be applied (typically scikit-learn linear regression [7]). Then, the predictions can be retrieved using the newly determined model. Finally, the predictions can be evaluated to retrieve a fitness score and returned along with the model. These functions can be seen in Algorithms 3 and 5.

H. Program Flow

The main algorithm follows a general program flow for evolution. The program flow is as follows:
First, a random population is created and evaluated. Then, the main generational loop is entered. This loop will iterate from generation 0 until the user-specified maximum generation. Within the loop, the following steps are performed. Select elite individuals, and add them to the new population. Then, create n-eliteCount offspring using tournament selection and crossover. Following the offspring creation, they should be mutated using one of the mutation functions described above. Finally, the mutated individuals should be added to the new population with the elite individuals. This population should now be evaluated. Pseudocode can be found in Algorithm 1.

III. User Guide

This MGGP system is designed for beginner to advanced users, providing support for easy pickling, allowing the user

Algorithm 1 Genetic Program Flow

```
population ← initializePopulation()
evaluatePopulation(population)
for gen = 0 to maxGen do
  elites ← population.selectBest(eliteCount)
  parents ← [(populationSize − eliteCount)]
  for i = 0 to parents.length do
    parents[i] ← tournamentSelection(population)
  end for
  offspring ← [(populationSize − eliteCount)]
  for i = 0 to parents.length do
    if random.double < crossoverProbability then
      offspring1, offspring2 ←
      crossover(parents[i], parents[i + 1])

      offspring.append(offspring1)
      offspring.append(offspring2)
      i++
    else
      offspring.append(parent)
    end if
  end for
  for Individual ind in offspring do
    if random.double < mutationProbability then
      mutate(ind)
    end if
  end for
  new_population ← [ ]
  new_population.append(elites)
  new_population.append(offspring)
  population ← new_population
  evaluatePopulation(population)
end for
```

to easily multi-thread each run, or potentially the fitness function. To get started the provided toolbox must imported into your Python application which handles the kernel logic to get your GP up and running swiftly. To begin create a Python environment using version 3.9 or later. Install all of the required dependencies (sklearn [7] and numpy [9]). Now, the kernel of the project should be set up and you can continue to build your GP application. There are some critical steps to ensuring this GP system works.

First, the fitness function must follow the following schema, the method signature must have the 3 parameters, individual, operators, and data. The function then should evaluate all genes on the data provided. Then using the evaluations and target data fit the linear model. This fit linear model can be used to generate predictions, and these predictions should be used to evaluate the fitness. Lastly, return the fitness and model used.

Secondly, after the fitness has been defined for a problem, custom language must be defined as functions.

Thirdly, the user must define the custom language, and a set of values for each parameter outlined in the parameter description must be determined.

Finally, the MGGP system can be run by passing in the parameters as the arguments, and the gp will have the following return values: [genAvgs, genMins, genMaxs, genMeds], best_individual, best_model, best_fitness.

IV. Experimental Setup (1)

A. Problem Statement

Symbolic regression describes the task of attempting to derive a function to fit a series of coordinate pairs. GP applications can excel at deriving these functions as GPs can search a wide variety of solutions while utilizing evolutionary principles to hone in on local optima. This first experiment seeks to compare the performance of a regular GP application developed using DEAP [1], with the performance of a MGGP application developed with the novel system.

B. MGGP System

The MGGP system was implemented as described in the system implementation. All relevant parameters for the MGGP system can be found in Table I. When supplying parameters to the MGGP system, care was taken to maintain as much similarity as reasonable between the GP parameters, and the MGGP parameters. For mutation, it was empirically determined that mutating each gene was most effective for this problem, so that mutation operator was used.

Parameter	Value
Population Size	**300**
Max Number of Genes	**4**
Number of Generations	**50**
Crossover Rate	**0.9**
Mutation Rate	**0.2**
Min Initial Size	**2**
Max Initial Size	**5**
Max Crossover Growth	**3**
Max Mutation Growth	**3**
Max Global Depth	**8**
Elitism Count	**1**
Tournament Size	**3**

TABLE I
PARAMETERS SUPPLIED TO THE MGGP SYSTEM

C. GP System

A modified version of the DEAP system [1] was used. Modifications were made to the DEAP system such that elitism was supported. This decision was made as previous experiments had determined a marginal improvement using elitism [4]. Elitism is the process of selecting the *n* best individuals, and carrying them over to the next generation without modifications. These individuals can still be selected by tournament selection to produce offspring. For this experiment, an elitism of 1 was chosen. All other parameters for this system can be found in Table II.

Parameter	Value
Population Size	**300**
Number of Generations	**50**
Crossover Rate	**0.9**
Mutation Rate	**0.2**
Min Initial Size	**2**
Max Initial Size	**5**
Max Size After Crossover	**17**
Max Size After Mutation	**17**
Elitism Count	**1**
Tournament Size	**3**

TABLE II
PARAMETERS SUPPLIED TO DEAP

1) Initialization Method

The DEAP system used a full tree generation method whenever initialization is required. This method was used initially to generate a random population, and this method is used in mutation to generate a new sub-tree from a specific point. This function will generate a tree with a size between the minimum initial size and the maximum initial size.

2) Selection Operator

The DEAP system used tournament selection for the selection operator. This operator was used when determining parents for crossover.

3) Crossover Operator

The DEAP system used a one-point crossover operator, where an individual node was selected within each individual. These nodes are directly exchanged between individuals.

4) Mutation Operator

The mutation operator supplied to DEAP was dependent on the initialization method. This mutation operator would select a node randomly within an individual, and initialize a new sub-tree from that node, with a height between the minimum initial size, and the maximum initial size.

D. Systems Language

The language supplied to both the MGGP and GP system was the same to ensure fairness between tests. The language is designed to be sufficient enough such that the GP systems are able to reproduce the target function. This language is composed of standard mathematical operators as defined in Table III. For this problem, the only mathematical operator that is not standard is protected division.

Function	Arity	Example Usage
Add	2	$x + y$
Subtract	2	$x - y$
Multiply	2	$x \times y$
Protected Division	2	$\frac{x}{y}$ (1 if $y \equiv 0$)
Cosine	1	$\cos(x)$
Sine	1	$\sin(x)$
Tangent	1	$\tan(x)$

TABLE III
FUNCTION DEFINITIONS

1) Protected Division

Protected Division is an operator that was added to guard against divide-by-zero errors. This function works identically to division, however, if divide-by-zero is attempted, 1 is returned.

E. Terminals & Ephemerals

The terminals and ephemerals supplied for this problem are relatively simplistic. As the goal is to evolve an expression based on the x-axis, the x-coordinate is supplied as a terminal, while a random float between -1 and 1 is used for ephemeral value. These values are described in Table IV

Terminal/Ephemeral	Type
x Coordinate	**Float**
Random (-1 to 1)	**Float**

TABLE IV
TERMINALS SUPPLIED TO BOTH GP SYSTEMS

F. Data

The data for this experiment was relatively simplistic, the independent variable supplied to the systems was the x-coordinate. This independent variable was used to predict the dependent variable, the y-coordinate. X-coordinates supplied were from -1 to 0.9 in intervals of 0.1, supplying 20 total points. Y-coordinates were supplied as a function of the x-coordinate. This target function is defined as follows:

$$y = \frac{\sin(x) \cdot \cos(x)}{\tan(x) + 1} + \left(x^3 - 2x^2 + 5x - 3\right) \cdot \sin(x) \cdot 0.5$$

G. Fitness Heuristic

The fitness heuristics for this problem were relatively simple to implement and each system follows the same concept. Each fitness function will evaluate the individual on all the points, and return the MSE. The MGGP fitness function adds on to the original fitness function by fitting the multigene individual to the data. Fitness functions have been provided in Algorithm 2 and Algorithm 3.

Algorithm 2 GP Symbolic Regression Fitness

function SYMBOLICGPFITNESS(individual, points)
  $func \leftarrow$ toolbox.compile(expr $= individual$)
  $sqerrors \leftarrow$ ((func(x) - targetFunc(x))^2 for x in $points$)
  $summed \leftarrow$ math.fsum$(sqerrors)$
  $average\_MSE \leftarrow \frac{summed}{\text{len}(points)}$
  return $average\_MSE$
end function

H. Experiment Performed

The experiments were performed as described in the experimental setup on both the regular GP system, as well as the MGGP system. These experiments were designed to be as similar as possible to try and best isolate the systems themselves. To collect more statistically significant results, runs

Algorithm 3 MGGP Symbolic Regression Fitness

```
function MGGPFitness(individual, ops, data_points)
    X ← np.array([
    for data in data_points do
        [
        for gene in individual do
            gene.evaluate(data, ops)
        end for
        ]
    end for
    ])
    y ← np.array([data['actual'] for data in data_points])
    model ← LinearRegression()
    model.fit(X, y)
    predictions ← model.predict(X)
    mse ← np.mean((predictions − y)^2)
    return mse, model0
end function
```

were performed across 10 unique seeds for each experiment, and the graphs represent the average of this data.

V. Results (1)

The results of this experiment show various improvements within the MGGP model. Firstly, when comparing Figures 1 and 4 one can notice huge spikes in the regular GP graph while they do not appear in the MGGP graph. This is because the mathematical weighting function is able to help fit the expressions evolved by the MGGP functions, preventing it from generating huge outlier solutions. In contrast, the regular GP model can generate a single value so large, it skews the average and makes the graph relatively unreadable. When comparing Figures 2 and 5, it becomes evident the GP systems converge at a relatively similar rate for this problem; however, the MGGP system starts with a much lower fitness, likely due to the mathematical weighting applied to each individual. Comparing Figures 3 and 6 can showcase the advantages of MGGP. The MGGP system again starts with a much better solution initially due to the mathematical weighting. However, the MGGP system can also evolve a good solution much faster than the regular GP system; evolving the solution approximately 20 generations earlier. Further results can be drawn from Table V. In this table, it becomes evident that for some problems, MGGP is able to evolve better solutions than a regular GP implementation. In this table, MGGP has an average best score of over 200x better than the average best score of the regular GP system. Furthermore, one can examine the best score of regular GP and the average best score of MGGP, when examining these values, it becomes apparent that MGGPs average best score is even significantly better than the overall best score of the regular GP system. Finally, when examining the best score produced by the MGGP system, one can notice that the best score is multiple orders of magnitude better than the score of the regular GP system.

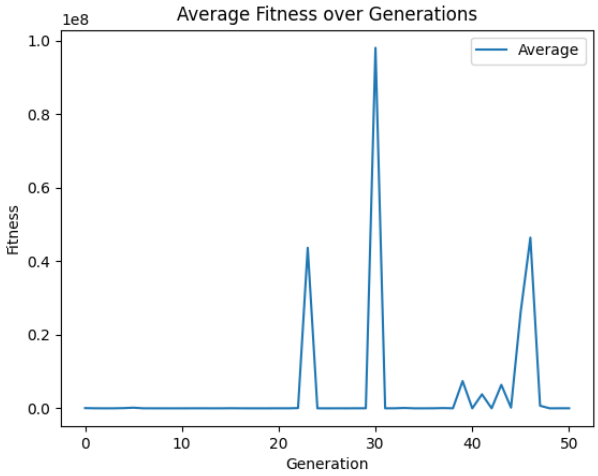

Fig. 1. GP Average Fitness Plot for Symbolic Regression

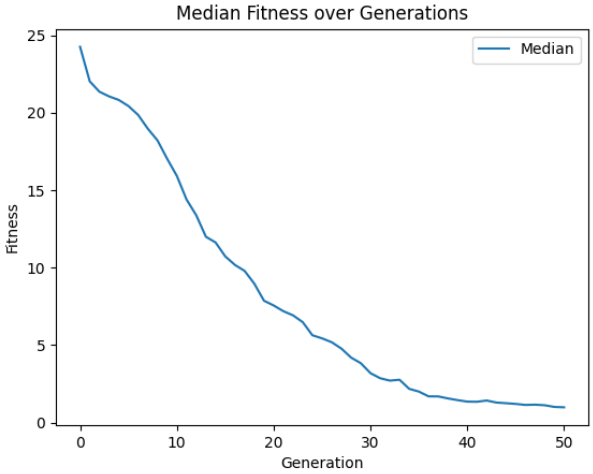

Fig. 2. GP Median Fitness Plot for Symbolic Regression

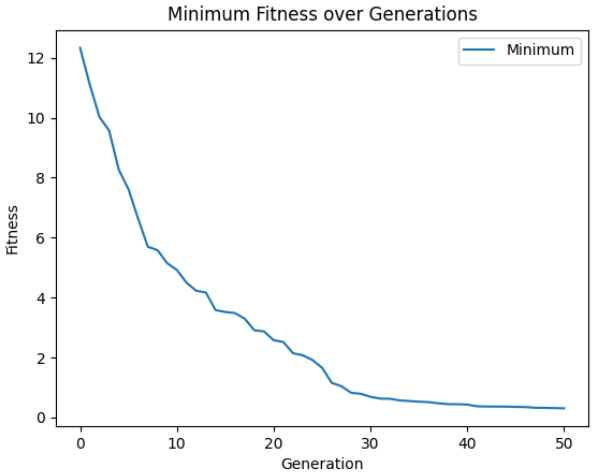

Fig. 3. GP Minimum Fitness Plot for Symbolic Regression

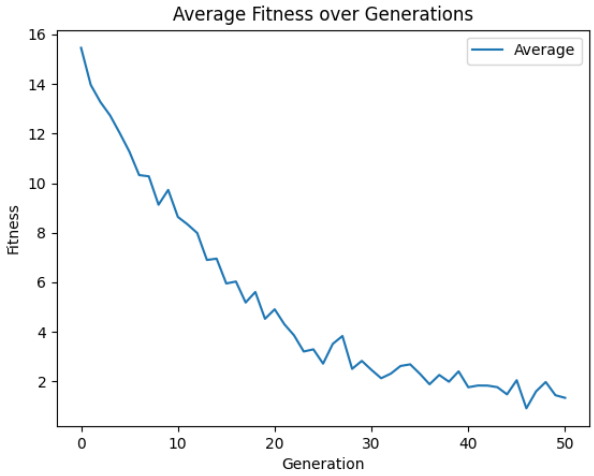

Fig. 4. MGGP Average Fitness Plot for Symbolic Regression

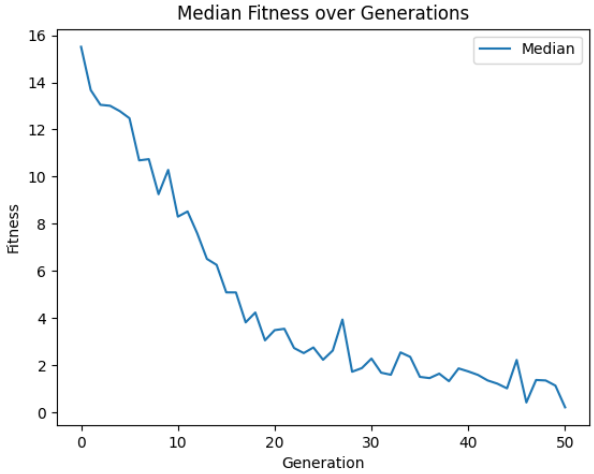

Fig. 5. MGGP Median Fitness Plot for Symbolic Regression

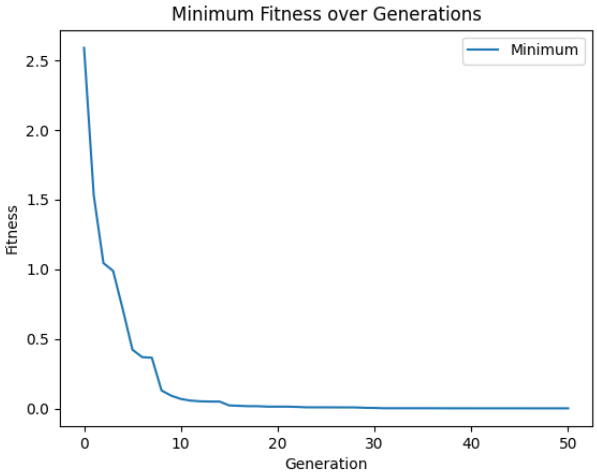

Fig. 6. MGGP Minimum Fitness Plot for Symbolic Regression

System	Average Best Score	Best Score
GP	0.3031	0.0105
MGGP	0.0015	$9.4100 * 10^{-7}$

TABLE V
COMPARISON OF AVERAGE BEST SCORES AND BEST SCORES FOR SYMBOLIC REGRESSION

VI. EXPERIMENTAL SETUP (2)

A. Problem Statement

The UCI Machine Learning Repository provides various datasets for training and testing ML models. One of these datasets describes a rice classification [6]. This problem involves analyzing seven features of a grain of rice, and using these features to predict which species the grain of rice is; Cammeo or Osmancik. GP-based systems can excel at solving this problem and have been proven historically to solve them with a high degree of accuracy [2]. For this reason, this ML problem is a good candidate problem to compare MGGP with a traditional GP system.

B. MGGP System

The MGGP system was implemented as described in the system implementation. All relevant parameters for the rice classification experiment can be found in Table VI. When determining parameters for the MGGP system, care was taken to ensure that similarity between the MGGP system and the GP system existed, while the parameter choice for the MGGP system was still reasonable. It was empirically determined the most effective mutation operator was the operator that mutated each gene. For this reason, this mutation operator was used.

Parameter	Value
Population Size	**400**
Max Number of Genes	**5**
Number of Generations	**50**
Crossover Rate	**0.9**
Mutation Rate	**0.3**
Min Initial Size	**2**
Max Initial Size	**3**
Max Crossover Growth	**2**
Max Mutation Growth	**2**
Max Global Depth	**4**
Elitism Count	**1**
Tournament Size	**3**

TABLE VI
PARAMETERS SUPPLIED TO THE MGGP SYSTEM

C. GP System

The DEAP system [1] was modified to incorporate elitism for this experiment as described in the first experiment, These modifications were made as elitism was determined previously to have a positive impact on the evolutionary process [5]. For this experiment, an elitism of 1 was chosen meaning the single best individual in each generation will be copied to the next generation without modification. All relevant parameters for the GP system can be found in Table VII.

Parameter	Value
Population Size	**400**
Number of Generations	**50**
Crossover Rate	**0.9**
Mutation Rate	**0.3**
Min Initial Size	**2**
Max Initial Size	**3**
Min Mutation Growth	**0**
Max Mutation Growth	**2**
Max Size After Crossover	**17**
Max Size After Mutation	**17**
Elitism Count	**1**
Tournament Size	**3**

TABLE VII
PARAMETERS SUPPLIED TO THE DEAP SYSTEM

1) Initialization Method

To initialize the trees, the DEAP system generated a full tree. This method was used to generate the random population required for the evolutionary process. This function generates a tree with a size between the specified minimum and maximum sizes.

2) Selection Method

The DEAP system was supplied with the built-in tournament selection operator for selection. This operator was utilized to determine parents for the crossover process.

3) Crossover Method

The DEAP system was supplied with a built-in one-point crossover operator. This function selects a point within each parent and exchanges the sub-trees at those points.

4) Mutation Method

To mutate individuals, the DEAP was supplied with a built-in mutation operator, along with a new generate tree function. This was done to allow the mutation to generate trees of a slightly smaller size than the initial trees. The mutation operator worked by selecting a point within an individual and generating a new sub-tree with a depth between the specified minimum and maximum depth.

D. Systems Language

The language supplied to the MGGP and the GP system was identical to ensure fairness in the experiment. The language was designed to be sufficient enough to generate a working model, without being too verbose. This language is composed of standard mathematical operators and two custom functions, protected division and protected log. This language is described in Table VIII.

Function	Arity	Example Usage
Add	2	$x + y$
Subtract	2	$x - y$
Multiply	2	$x \times y$
Protected Division	2	$\frac{x}{y}$ (1 if $y \equiv 0$)
Protected Log	1	$\ln(x)$ (0 if $x \leq 0$)
Sine	1	$\sin(x)$
Cosine	1	$\cos(x)$

TABLE VIII
FUNCTION DEFINITIONS

1) Protected Division

Protected Division is an operator that was added to guard against divide-by-zero errors. This function works identically to division, however, if divide-by-zero is attempted, 1 is returned.

2) Protected Log

Protected Log is an operator that was added to protect against arithmetic errors when taking the logarithm of a number ≤ 0. When the input ≤ 0, this function will return 0. Otherwise, this function will return the natural logarithm of the input.

E. Terminals & Ephemerals

The terminals supplied to the GP system were all the features of the rice grain, except for the classification, as well as a random float ephemeral between -1 and 1. Extra care was taken to ensure both GP systems were not supplied with the classification data for evolution as that would make the problem trivial. The terminals and ephemerals are described in Table IX.

Terminal/Ephemeral	Type
Area	**Float**
Perimeter	**Float**
Major_Axis_Length	**Float**
Minor_Axis_Length	**Float**
Eccentricity	**Float**
Convex Area	**Float**
Extent	**Float**
Random (0-1)	**Float**

TABLE IX
TERMINALS SUPPLIED TO BOTH GP SYSTEMS

F. Data

The data supplied to both GP systems was from the UCI Machine Learning Repository [6]. This data described 7 floating point features about the grain of rice as well as an 8th feature which was a species classification for the grain of rice. This column was iterated over, values for "Cammeo" were replaced with 0, and values for "Osmancik" were replaced with 1. Following this, the data was shuffled and then split into training and testing data. The data contained 3810 rows, 450 rows were used for training, and 3360 rows were used for testing. The class variable, would be the dependent variable while the other 7 terminals are considered independent variables.

G. Fitness Heuristics

The fitness functions for both GP systems revolved around minimizing misses. The misses are when the GP incorrectly classifies a data point as the incorrect class. For both fitness functions the individuals were evaluated on the entire training set and the count of missed data points was returned. If the prediction function predicted > 0.5, this prediction is assumed to be an Osmancik grain, otherwise, the prediction is assumed to be a Cammeo grain. The fitness function for the DEAP GP

system was defined in Algorithm 4 and the fitness function for the MGGP system was defined in Algorithm 5.

Algorithm 4 Evaluate Rice

```
function EVALRICE(individual, points, trainingAnswers)
    func ← toolbox.compile(expr = individual)
    misses ← 0
    for i ← 0 to len(points) do
        classification ← func(*points[i])
        if classification ≤ 0.5 then
            if trainingAnswers[i] == 1 then
                misses ← misses + 1
            end if
        else
            if trainingAnswers[i] == 0 : then
                misses ← misses + 1
            end if
        end if
    end for
    return misses,
end function
```

Algorithm 5 Multi-Gene Fitness Function

```
function MGGPRICEFITNESS(individual, ops, data_points)
    misses ← 0
    X ← [ ]
    Y ← [ ]
    for data in data_points do
        X.append([gene.evaluate(data, ops) for gene in individual])
        Y.append(data['Class'])
    end for
    X ← np.array(X)
    Y ← np.array(Y)
    model ← LinearRegression()
    model.fit(X, Y)
    for i ← 0 to len(X) do
        predictedY ← model.predict([X[i]])
        if predictedY < 0.5 then
            if Y[i] == 1 then
                misses ← misses + 1
            end if
        else
            if Y[i] == 0 then
                misses ← misses + 1
            end if
        end if
    end for
    return misses, model
end function
```

H. Experiment Performed

The experiments were performed following the procedures outlined in the experimental setup. These experiments were completed over 10 runs using unique seeds to produce a more statistically significant result. At the conclusion of every run, the best solution for each run was evaluated and a confusion matrix was generated.

VII. Results (2)

This experiment also shows improvements within the MGGP system over the regular GP system, but in different metrics than from the symbolic regression experiment. When comparing Figures 7 and 8 from the regular GP system with Figures 10 and 11 from the MGGP system, it becomes apparent that the MGGP system again starts with a significantly better fitness score, and this time converges much faster than the regular GP system. Figures 9 and 12 showcase the MGGP model reaching a good solution significantly faster than the regular GP model. These figures also showcase the strength of the weighting functions as the initial fitness score of the MGGP model is drastically lower than that of the regular GP model. When examining Table X, the average best scores can be compared between the MGGP and GP systems. These scores show that on average the MGGP systems have some degree of improvement over the GP systems, however, the best scores show that out of the 10 runs performed, the best results from the MGGP and GP systems perform similarly. The confusion matrices found in Tables XI and XII support this data.

System	Average Best Score	Best Score
GP	89.15%	93.07%
MGGP	92.61%	93.04%

TABLE X
COMPARISON OF AVERAGE BEST SCORES AND BEST SCORES FOR RICE CLASSIFICATION

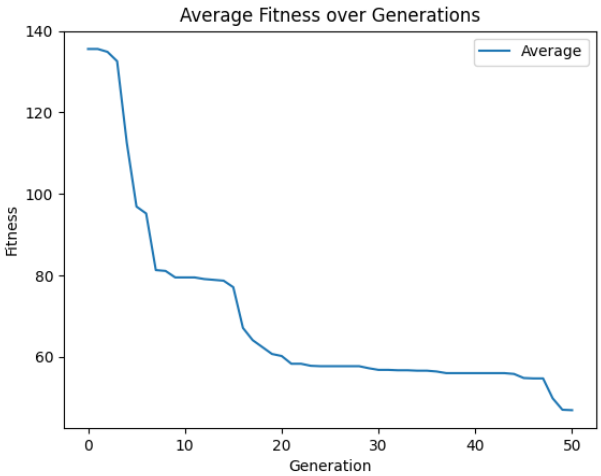

Fig. 7. GP Average Fitness Plot for Rice Classification

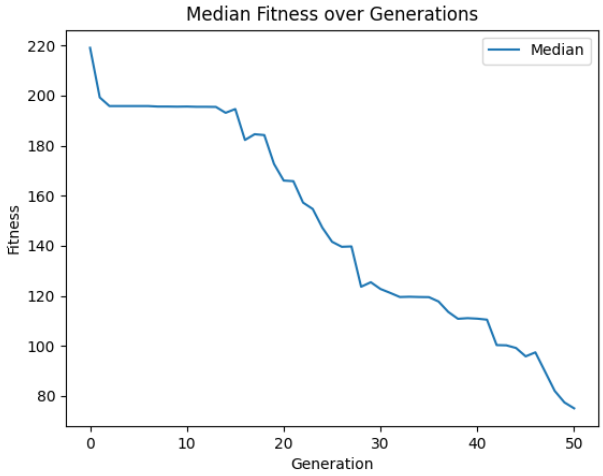

Fig. 8. GP Median Fitness Plot for Rice Classification

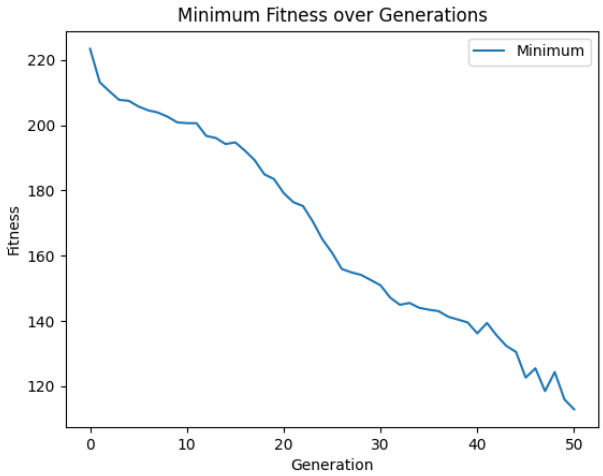

Fig. 9. GP Minimum Fitness Plot for Rice Classification

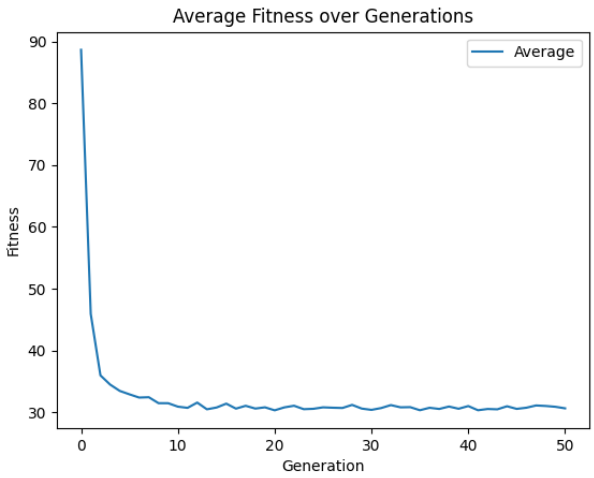

Fig. 10. MGGP Average Fitness Plot for Rice Classification

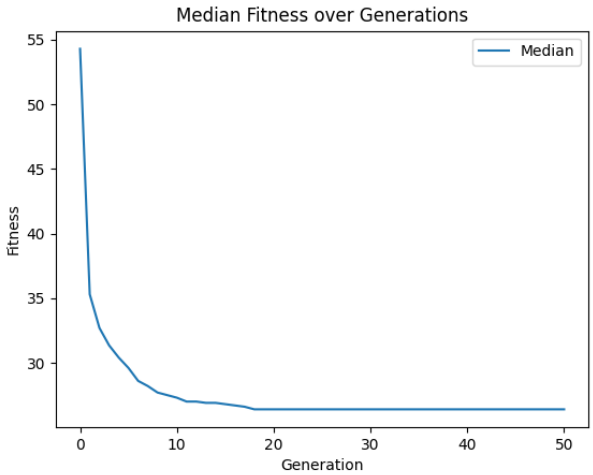

Fig. 11. MGGP Median Fitness Plot for Rice Classification

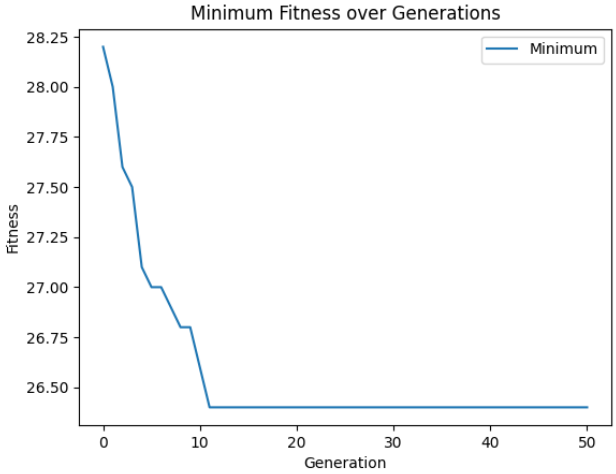

Fig. 12. MGGP Minimum Fitness Plot for Rice Classification

		Predicted	
		Osmancik	**Cammeo**
Actual	**Osmancik**	93.93%	6.07%
	Cammeo	8.09%	91.91%

TABLE XI
CONFUSION MATRIX FOR GP

		Predicted	
		Osmancik	**Cammeo**
Actual	**Osmancik**	93.21%	6.79%
	Cammeo	7.20%	92.80%

TABLE XII
CONFUSION MATRIX FOR MGGP

VIII. Conclusion

This report outlines the issues with the accessibility of MGGP systems, and the steps taken to successfully implement a novel MGGP system written in Python in an attempt to resolve this issue. A user guide has been provided as well to help users get started with the system and provide them with what information is required to use the GP system in a clear and concise manner. The performance of the MGGP is then determined; establishing that MGGP can provide benefits in many areas depending on the problem. Possible improvements include convergence rates, initial scores, generations required to find a good solution, and even possible improvements in final scores. Further work should be done to both refine the GP system and identify further benefits of MGGP, this work could include extending the MGGP system to include other selection, crossover and mutation operators, or this could include testing different problem types to identify other advancements of MGGP. MGGP could also be tested with early termination criteria to end a run early. This has the potential to speed up the evolutionary algorithm significantly by removing unnecessary evaluations. For applications with very large datasets, large tree depths, large terminal sets, or a high number of runs, the computation time saved could make MGGP more sustainable and feasible in energy savings and time investment respectively, which could attract investment opportunity and incentivize use in business or research.

References

[1] DEAP Documentation. https://deap.readthedocs.io/en/master/

[2] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. *A Field Guide to Genetic Programming*. 2008.

[3] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*.

[4] Corbett, C & Carvalho, C (2024). *Genetic Programming in Symbolic Regression Problems*.

[5] Corbett, C & Caralho, C (2024). *Genetic Programming in Rice Classification Problems*.

[6] UCI Machine Learning Repository dataset. https://archive.ics.uci.edu/dataset/545/rice+cammeo+and+osmancik

[7] scikit-learn Linear Regression Weighting Documentation. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

[8] GPTIPS: an open source genetic programming toolbox for multigene symbolic regression Searson, D.P., Leahy, D.E. & Willis, M.J. Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 (IMECS 2010), Hong Kong, 17-19 March, 2010.

[9] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585, 357–362. https://doi.org/10.1038/s41586-020-2649-2

APPENDIX

This appendix contains evolved expressions showcasing both MGGP individuals and GP individuals.

A. Experiment 1

1) GP

Best evolved expression:

$$y = ((sin((x*((cos((-0.9311955137342252))*((-0.950062020051504)+0.14931956605807994))-(x*(-0.30175378291087407)))))$$
$$+((x+(0.1600638272459014-0.6660196922119248))*tan(x)))-((sin(((x-(cos(tan((x-(cos(cos(cos(x)))+$$
$$(cos(cos((0.26188500225636857-0.5246779173115286)))*(cos((0.26188500225636857-0.5246779173115286))*$$
$$((x-(-0.4063136968692145))/(x*(-0.30175378291087407)))))))))+sin(((((tan(0.640030684062628)*tan(x))$$
$$-((0.12563693725781677/x)-cos(x)))-((tan((-0.9525492228566996))-(x*0.7311774554116763))+tan(((-0.8932381650998957)$$
$$*0.2163993421621211))))*(tan(((x/x)+x))-(((x-0.5107379143613104)/sin(x))+sin(((-0.053454176922636876)+x)))))))$$
$$-sin((sin((sin(sin((x/x)))*sin((-0.9188750578712725))))*cos((sin(0.7658535018438437)-(x-0.6224198814499777)))))$$
$$-tan((x+x)))*((sin(x)*sin(x))*cos((x-(cos(cos(cos(x)))+(cos((0.26188500225636857-0.5246779173115286))*$$
$$(cos((-0.30175378291087407))*((x-(-0.4063136968692145))/(x*(-0.30175378291087407))))))))))$$

2) MGGP

Each non-indented line represents a single gene in the individual.

Best evolved expression:

$y = -4.830768970417401$
$-2.658902155076137*sub(sin(x), add(0.8621482912806118, mul(div(add(0.368440251108886, x), 0.798324629284783), x)))$
$+1.9377966338330816*sub(cos(x), add(cos(cos(sub(x, -0.6689799018498213))), x))$
$+0.716593351226182*add(add(add(x, cos(0.6756739024852232)), div(sub(x, x), 0.7387702487547847)),$
$mul(div(x, add(cos(x), sin(x))), mul(x, sin(-0.9270332598323752))))$
$-1.504484642334627*sub(div(0.6852288795237669, sin(sub(-0.35876494713435814, sin(sin(0.9463307739729208))))),$
$add(0.16322307465846575, sin(add(x, x))))$

B. Experiment 2

1) GP

Best evolved expression:

$prediction = (Major_Axis_Length/log(cos(log(cos((log(Major_Axis_Length)/0.11586865878832175))))))$

2) MGGP

Best evolved expression:

$prediction = -3.8461703061244266$
$+646.4307794056832*sin(div(Minor_Axis_Length, Convex_Area))$
$+0.016320198639018277 * sin(add(Perimeter, Extent))$
$+0.09421907911318467 * cos(cos(Minor_Axis_Length))$