HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT REPORT

SOLVING EQUATIONS USING GENETIC ALGORITHM

Course	**:**	**Project 1**
Course ID	**:**	**IT3910E**
Instructor	**:**	**Assoc. Prof. Tran Dinh Khang**
Members	**:**	

Student Name	Student ID	Email
Dinh Nguyen Cong Quy	20214927	quy.dnc214927@sis.hust.edu.vn
Nguyen Trung Truc	20214936	truc.nt214936@sis.hust.edu.vn
Phan Dinh Truong	20214937	truong.pd214937@sis.hust.edu.vn

Hanoi, December 2023

ACKNOWLEDGEMENT

We would like to express our deep gratitude to those who contributed and supported us during the implementation of this project.

We are especially grateful to Assoc. Prof. Tran Dinh Khang for his unwavering guidance, invaluable insights, and suggestions that significantly enhanced our comprehension of the Genetic Algorithms domain and facilitated the successful completion of this project.

Additionally, we express our appreciation to our fellow teammates whose enthusiastic contributions and strong collaboration were instrumental in bringing this project to fruition. The collective contributions of everyone involved have been indispensable in attaining commendable outcomes for this project.

With sincere thanks,

ABSTRACT

Our project focuses on developing a program using Python programming language to solve algebraic equations through the application of Genetic Algorithms. This program gives users the ability to enter the equation and automatically find the solution of that equation. We have developed a user-friendly interface that allows users to enter their equations and monitor the optimization process through the Genetic Algorithm. While the program is running, users have the ability to monitor graphs showing the change of the variable and the value of the function.

Group of Students

Table of Contents

1. Introduction

In the area of computational problem-solving, Mathematical models for a wide variety of problems in science and engineering can be formulated into equations of the form:

$$f(x) = 0 \qquad (1)$$

The manipulation and resolution of (1) represent a cornerstone challenge. The ability to parse complex equations from strings of symbols and subsequently optimize their solutions holds profound importance across scientific, engineering, and technological domains.

There exist a great variety of methods to find the roots of ***(1)***, including analytical methods [1], graphical methods [2], trial and error methods [3], … While these methods have proven effective for many scenarios, their applicability can be constrained by the complexity of equations or the computational resources required. Equations in real cases often demand innovative approaches that transcend the limitations of traditional methodologies.

In this project, we propose an approach to solve two main problems:

- Convert algebraic expressions represented as a string into a form that can be easily reused for computing the result of the expressions,
- Apply the Genetic Algorithm (GA) for finding a root of the given equation.

It is worth mentioning that this project focuses mainly on the parsing and solving of real elementary single-variable equations, and in the case of multiple-root, the developed system will output only one real fittest solution, thus the results may be variant at different runs.

The remainder of the paper is organized as follows. Section 2 reviews the background knowledge and states our approach to each particular problem. Section 3 presents the visualization of our method's result. Section 4 analyzes and discusses the experiments performed. Finally, conclusions are drawn in Section 5.

2. Methods

2.1. Input handling

The user's input is literally a string representation of an equation, which contains the equal sign ("=") to separate the two sides of the equation. Thus, before applying the parsing and evaluating process, we split the input equation into 2 parts: left-hand side (LHS) and right-hand side (RHS), then we perform conversion on both sides and calculate the subtraction of RHS from LHS. This output will be used for the fitness evaluation discussed in the later sub-section.

The process of evaluating the expression consists of 2 main phases: **Analyzing the user's input** and **Calculating the output**.

2.1.1. Analyzing the Input

The input string is firstly grouped into small segments called tokens. A token is an object that has a type and a value. For example, the characters in the input 12 + 24 are grouped into the tokens NUMBER:12, PLUS, and NUMBER:24. Through a section of code called Lexer, the input is converted into a stream of tokens with each token assigned to its corresponding type. Whitespaces are ignored by this lexer.

The token stream is handled by the parser, which will analyze the sequence of tokens to determine what is intended to happen and in what order, similarly to how we make sense of sentences based on the sequence and types of words. The rule on which the parser uses to determine the order is called grammar rule. In this project, the grammar rule is simply the basic mathematical order of precedence, which is implemented with the branching structure in the code.

From the stream of tokens combined with the grammar rule, the parser is responsible for building an intermediate representation data structure to store all the tokens hierarchically. In this project, the Abstract Syntax Tree (AST) is suitable for this requirement.

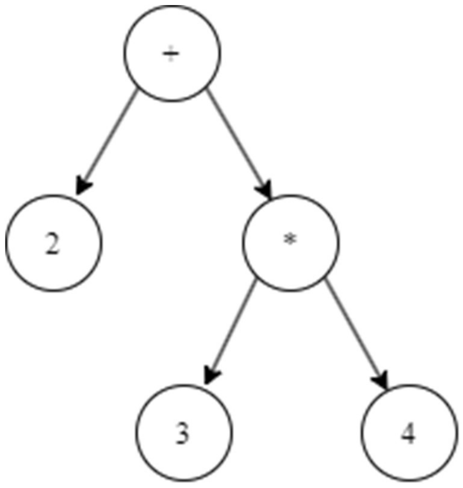


*Figure 1. AST for the expression 2 + 3 * 4*

The **AST** represents the abstract syntactic structure of a language construct where each interior node and the root node represents an operator, and the children of the node represent the operands of that operator [4]. Compared to a normal parser tree, **AST** is much more compact since it does not require to represent every detail from the real syntax, such as the parentheses, and the lower the height of a node, the higher priority it is in the expression. The implementation of **AST** will optimize the memory cost for storage and reduce the computational time for the traversal task that needs to be performed to find the expression 's result.

2.1.2. Calculating the Output

After constructing the **AST** representation from the parsed tokens, the evaluation of the expression takes place. This process involves recursively traversing the **AST** according to the defined mathematical operations and their precedence. Note that before the commencement of the evaluation process, any variables or functions present in the expression will be substituted by their values or resolving their definitions as required.

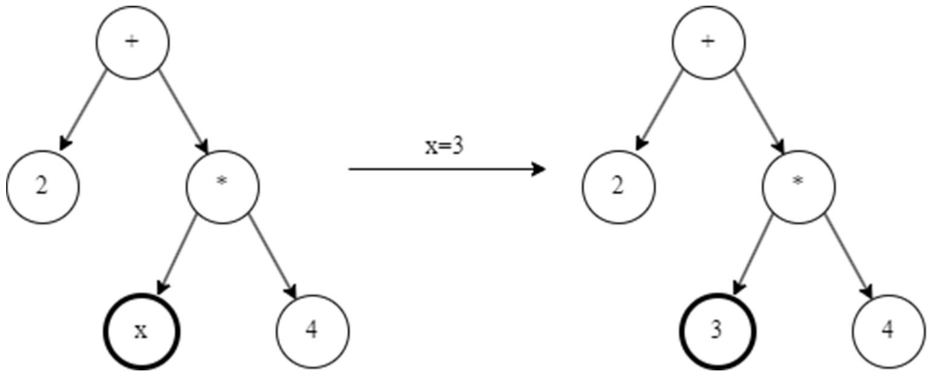


Figure 2. Subtitution of variable nodes

Beginning from the root node of the **AST**, the evaluation algorithm follows a depth-first approach, visiting each node and performing the corresponding operation based on the operator at that node. For instance, starting from the lowest precedence operations (typically addition and subtraction) and progressing upwards to higher precedence operations (such as multiplication, division, etc.), the algorithm computes the result by operating on the evaluated operands.

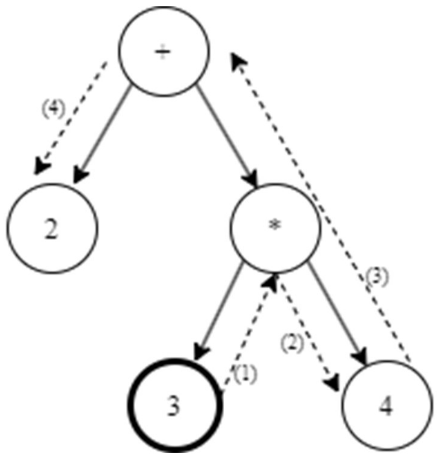


Figure 3. Traversing the AST to get the evaluated expression

During this recursive traversal, each operator node applies its operation to the results obtained from evaluating its child nodes. This step continues until the entire AST has been traversed, culminating in the computation of the final result of the expression.

2.2. Genetic Algorithm

Genetic Algorithm (GA) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic Algorithms are based on an analogy with genetic structure and behavior of chromosomes of the population. Following is the foundation of GA based on this analogy:

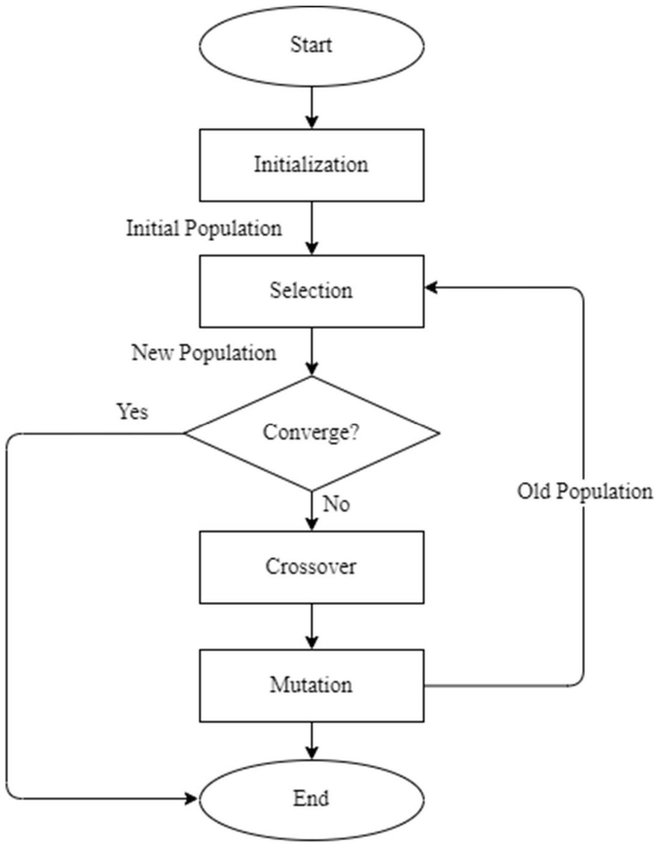


Figure 4. General schema of Genetic Algorithm

- Individual in population compete for resources and mate.
- Those individuals who are successful (fittest) then mate to create more offspring than others.
- Genes from "fittest" parents propagate throughout the generation, that is sometimes parents create offspring which are better than either parent.
- Thus each successive generation is more suited for their environment.

Search space: The population of individuals are maintained within search space. Each individual represents a solution in search space for a given equation. Each individual is coded as a 32-bit binary number (each bit is analogous to a gen of a chromosome).

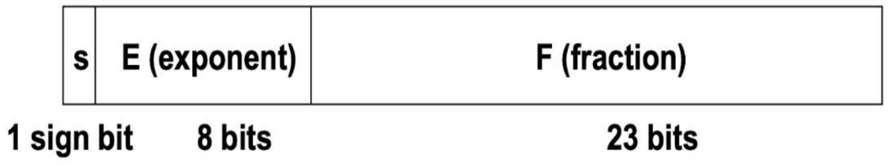


Figure 5. Encoding individual to 32-bit signed float (Source: internet)

The value of each individual is represented as floating point number as followed.

$$value = (-1)^{sign} \times 1.F \times 2^{E-bias}$$

Take an example: X = 1100 0001 0101 0110 0000 0000 0000 0000

$$sign = 1 \rightarrow X\ is\ negative$$

$$E = 1000\ 0010$$

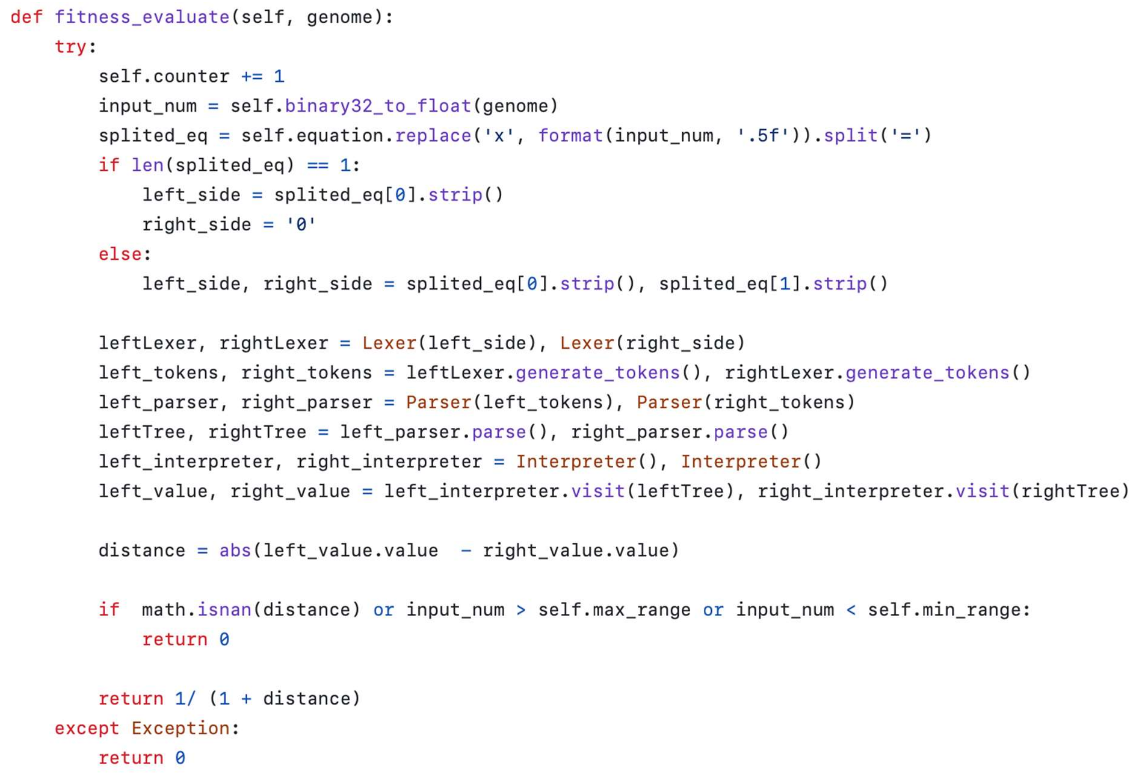
$$F = 10101100 \ldots 00$$

$$\rightarrow X = (-1)^1 \times 1.101011000 \ldots 00 \times 2^{130-127}$$

$$= -1.101011 \times 23 = -1101.011 = -13.375$$

Fitness score: A fitness score is given to each individual which shows the ability of an individual to “compete” [5]. The individual having optimal fitness score (or near optimal) are sought.

In this case, we take the inverse of the difference between the left hand side and the right hand side of an equation as its fitness score. The higher fitness score an individual has, the more likely it is to be chosen.



```python
def fitness_evaluate(self, genome):
    try:
        self.counter += 1
        input_num = self.binary32_to_float(genome)
        splited_eq = self.equation.replace('x', format(input_num, '.5f')).split('=')
        if len(splited_eq) == 1:
            left_side = splited_eq[0].strip()
            right_side = '0'
        else:
            left_side, right_side = splited_eq[0].strip(), splited_eq[1].strip()

        leftLexer, rightLexer = Lexer(left_side), Lexer(right_side)
        left_tokens, right_tokens = leftLexer.generate_tokens(), rightLexer.generate_tokens()
        left_parser, right_parser = Parser(left_tokens), Parser(right_tokens)
        leftTree, rightTree = left_parser.parse(), right_parser.parse()
        left_interpreter, right_interpreter = Interpreter(), Interpreter()
        left_value, right_value = left_interpreter.visit(leftTree), right_interpreter.visit(rightTree)

        distance = abs(left_value.value  - right_value.value)

        if  math.isnan(distance) or input_num > self.max_range or input_num < self.min_range:
            return 0

        return 1/ (1 + distance)
    except Exception:
        return 0
```

Figure 6. Implementation of Fitness function

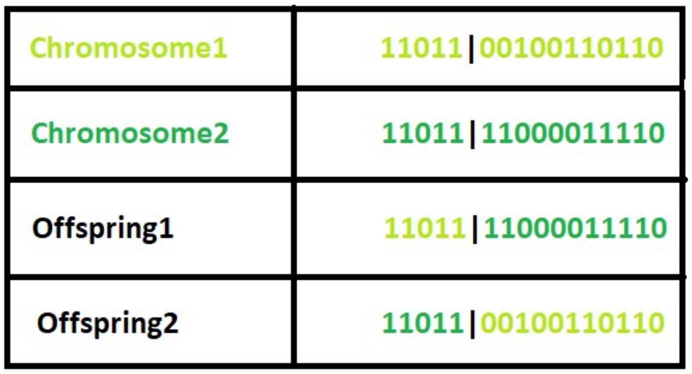
Main operators of Genetic Algorithm:

1. Randomly initialize populations: at first, we initialize **200 random solutions** for our given equation. Each individual in the population is represented by a gene string of **32 bits** in the length.
2. Determine the fitness score for each individual in the population.

$$difference = |LHS - RHS|$$

$$fitness\_score = \frac{1}{1 + difference}$$

3. Until convergence repeat:
   a. **Selection Operator**: Select 2 individuals with **best fitness scores** in a random group of individuals randomly selected from the population.
   b. **Crossover Operator**: Randomly select a slice point in each individual's gene. Since each individual in the population is represented by a string of **32 bits**, the slice point will be chosen between 1 to 31.
   The genes at these crossover sites of 2 individuals selected are exchanged thus creating a completely new individual (offspring).
   For example:



Chromosome1	11011|00100110110
Chromosome2	11011|11000011110
Offspring1	11011|11000011110
Offspring2	11011|00100110110

Figure 7. Single Point Crossover (Source: internet)

   c. **Mutation Operator**: For each bit in an individual's gene, perform a mutation with the probability of **mutation is 0.2**.
   If the random value is less than mutation_rate, change the bit value to the opposite bit in offspring to maintain the diversity in the population to avoid premature convergence.
   For example, $0 \rightarrow 1$ *or* $1 \rightarrow 0$ *in each bit in a genome with the mutation rate*
   d. **Calculate fitness** for the new population and updating the fitness score.

Initially, the equation will default to finding solutions in the range $\mathbf{-10^5}$ to $\mathbf{10^5}$. The user can customize this interval so that the convergence occurs faster.

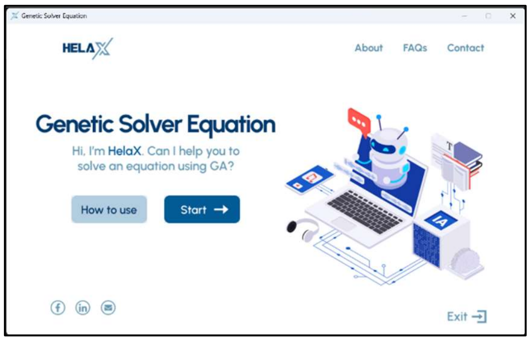
We **do not define the exact the number of generations.** Instead, we set the stopping condition by combining the conditions:

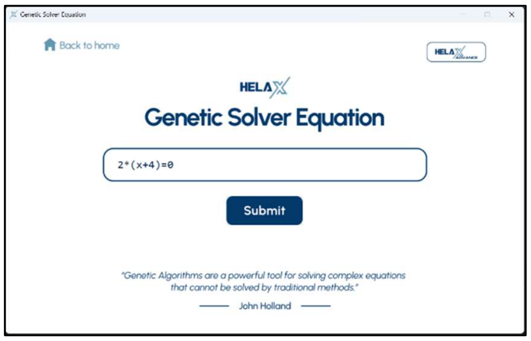
- Check the convergence of the solution by comparing the absolute value between the 2 closest solutions. If the convergence reached (with the difference less than $\mathbf{10^{-9}}$), this condition will be True.
- Check the fitness score. If the fitness score is greater than **0.999**, this condition will be True.

Moreover, to avoid non-convergence and infinite iteration, we limit the solution finding time to 10 seconds. If exceed, there is a high possibility that the equation has no solution because the fitness score is not convergence, or the fitness score cannot be improve any further.

3. Results

Display Graphic User Interface (GUI) allowing users to input equation. We have developed a user – friendly interface that allow users to enter equation easily.



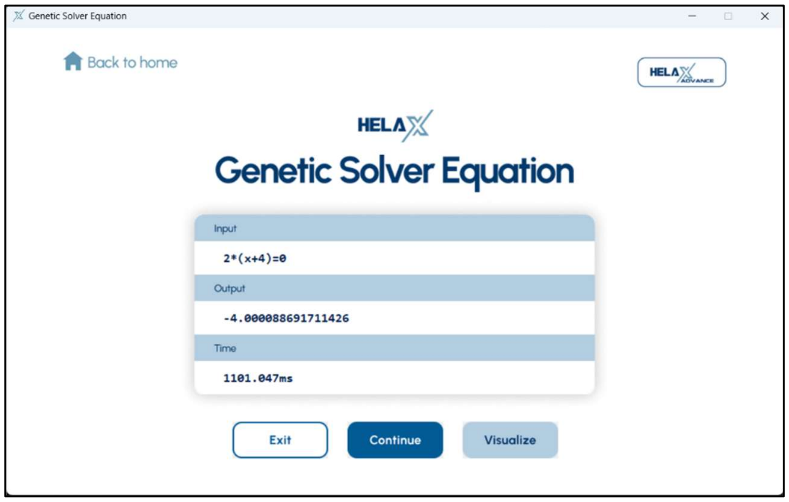


2*(x+4)=0

"Genetic Algorithms are a powerful tool for solving complex equations that cannot be solved by traditional methods."
— John Holland —

Figure 8. Program Menu and User Input

When user clicks the **"Submit"** button, the program will receive the input and use Genetic Algorithms to solve the equation. The solution of the equation and execution time will be displayed to the screen as follow:



Input
2*(x+4)=0
Output
-4.000088691711426
Time
1101.047ms

Figure 9. Solution Result of Equation

The user can track the solutions 'x' found by Genetic Algorithm through generations using graph by clicking the **"Visualize"** button:

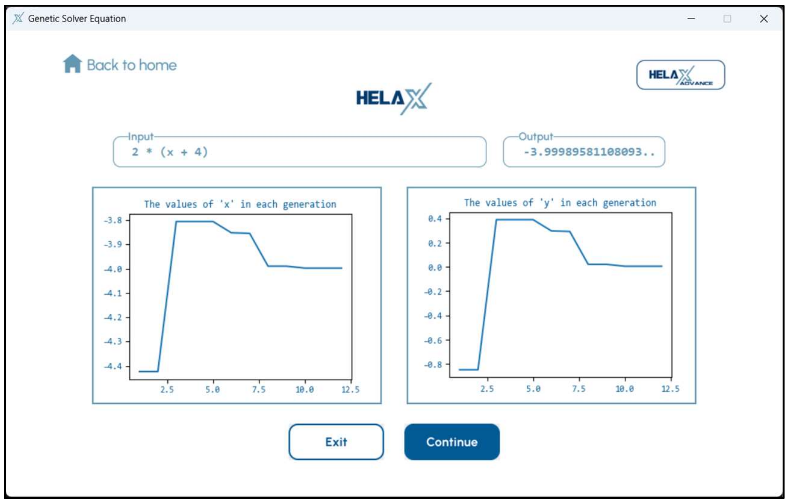


Figure 10. Visualization of x and f(x) in each generation

The graph is the value of the variables "x" and "y" in each generation. The line representing the value of "x" converges to solution value, and the value of "y" converges to the value of RHS (which default equals to 0) means that the equation has found a nearly exact solution.

If user want the faster convergence of f(x), they can change the interval to solve the equation by trying **HelaX Advance**:

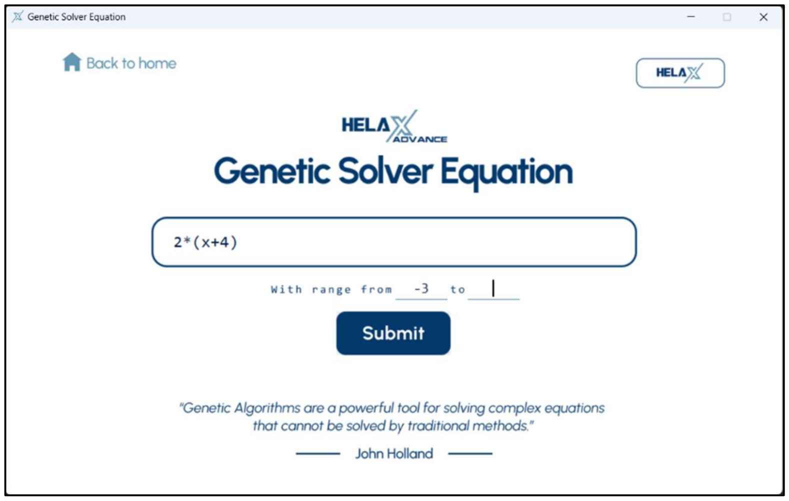
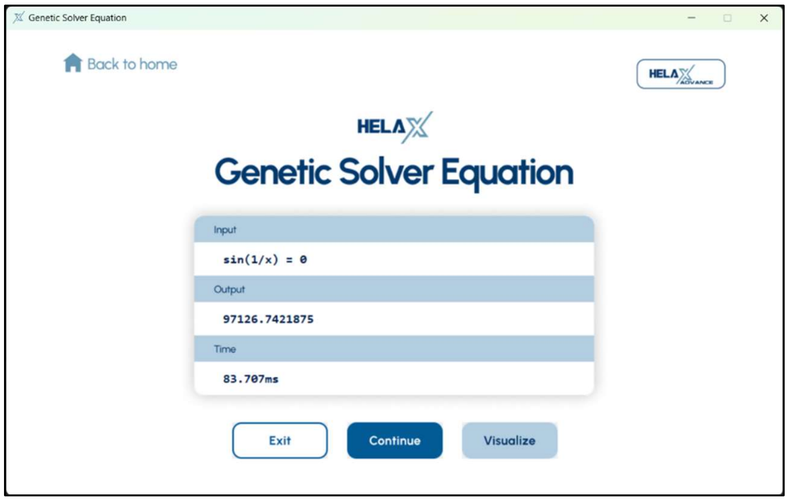


Figure 11. Change interval to solve equation

4. Discussion

In general, the developed system proved to be quite effective in finding a root of complex equations that can not be solved by traditional methods. The computational time is relatively fast (approximately 100-200ms) in the scenario of solvable equations. However, there exists some noticeable unsolved problems:

- **No solution condition:** In some cases, the given functions may reach 0 at infinity, but it is not necessarily equal to 0, For example, $\sin\left(\frac{1}{x}\right) = 0$ indeed doesn't have any real roots, but the system may output a large value as the answer, since the stopping conditions only concern searching time and improvement only.



```
sin(1/x) = 0
97126.7421875
83.707ms
```

Figure 12. Wrong Answer case

- **Slow convergence:** The speed of convergence relies heavily on the search range, thus in the worst case, if the solution tends to the upper limit, the algorithm may take much time to search for this solution.

5. Conclusion

While the **Genetic Algorithm** has proven adept at tackling single-variable equations, its application to multi-variable equations or systems with vector quantities requires further work. Moreover, enhancing the fitness evaluation and population initialization mechanism are crucial for reducing the search time of the algorithm. Additionally, refining the conditions for the case of ***"No Solution"*** is essential for better accuracy and broader application. Future advancements could involve improvement of the result visualization part, designing a more user-friendly interface, and adapting this framework to handle differential equations, opening doors to a wider array of mathematical challenges.

APPENDIX: Input Syntax

The following table shows the standard input syntax of the system

Function	Description
+, -, *, /	Add, Minus, Product, Divide respectively.
a ^ n	**a** to the power of **n**
e	Return Euler's number (2.7182...)
pi	Return PI (3.1415...)
sin(x)	Return the sine of a number **x**
cos(x)	Return the cosine of a number **x**
tan(x)	Return the tangent of a number **x**
cot(x)	Return the cotangent of a number **x**
log(x, base)	Return the logarithm of a number **x** to **base**. The default base is 10
arcsin(x)	Return the arc sine of a number **x**
arccos(x)	Return the arc cosine of a number **x**
arctan(x)	Return the arc tangent of a number **x**
arccot(x)	Return the arc cotangent of a number **x**
ln(x)	Return the natural logarithm of a number **x**
sqrt(x)	Return the square root of a number **x**
nroot(x, nth)	Return the n-th root of a number **x** (**n** > 1)
abs(x)	Return the absolute value of a number **x**
factorial(x)	Return the factorial of a number x (equal to **x**!)

REFERENCES

[1] Eberhart R., Simpson P. and Dobbins R. , *Computational Intelligence PC Tools*, AP Professionals, 1996

[2] J.Reeb. S.Leavengood, *Using the graphical method to solve system of linear equations*, Operation Research. October 1998

[3] E. Balagurusamy, *Numerical Methods*, Tata McGraw Hill, New Delhi, 1999

[4] Ruslan Spivak, *Let's Build a Simple Interpreter. Part 7: Abstract Syntax Tree*, Ruslan's Blog, December 2015

[5] Nikos E. Mastorakis, *Solving Non-linear Equations via Genetic Algorithms*, Hellenic Naval Academy, Terma Hatzikyriakou 18539, Piraeus, Greece, June 2005