

# Mathematical Function Reconstruction From a dataset using Genetic Programming

Submitted in the partial fulfillment of the requirements for  
the award of the degree of

## **BACHELOR OF TECHNOLOGY IN COMPUTER ENGINEERING**



UNDER SUPERVISION OF

Dr. Bashir Alam  
Assistant Professor

SUBMITTED BY

Abhishek Raj Chauhan (10-CSS-02)  
Prakhar Dhama (10-CSS-48)

DEPARTMENT OF COMPUTER ENGINEERING  
FACULTY OF ENGINEERING AND TECHNOLOGY  
JAMIA MILLIA ISLAMIA, NEW DELHI-110025  
YEAR: 2014

## **CERTIFICATE**

This is to certify that this project report (CEN-891) entitled “**Mathematical Function Reconstruction from a dataset using Genetic Programming**” is a bonafide work of Abhishek Raj Chauhan (10-CSS-02) and Prakhar Dhama (10-CSS-48) in partial fulfillment of Graduation degree of B. Tech Computer Engineering in Jamia Millia Islamia during the year 2014.

The project was carried out under my supervision. This project has not been submitted earlier for the award of any Degree or Diploma to the best of my knowledge and belief.

Date:



**Dr. Bashir Alam**

Assistant Professor

Dept. of Computer Engg.

Jamia Millia Islamia

New Delhi – 110025

**Dr. Tanvir Ahmad**

H.O.D.

Dept. of Computer Engg.

Jamia Millia Islamia

New Delhi – 110025

## **ACKNOWLEDGEMENT**

We are greatly indebted to our supervisor and guide Dr. Bashir Alam for his invaluable technical guidance, great innovative ideas and overwhelming moral support during the course of the project. This thesis could not have been written without his help. We thank him for his guidance, constant supervision and for providing us with the necessary resources to accomplish the given task within the time constraints.

We also express our gratitude to the Department Of Computer Engineering and the entire faculty members, for their teaching and guidance and encouragement. We are also thankful to our classmates and friends for their valuable suggestions and support.

**Abhishek Raj Chauhan (10-CSS-02)**

Dept. of Computer Engg.

Jamia Millia Islamia

New Delhi – 110025

**Prakhar Dhama (10CSS-48)**

Dept. of Computer Engg.

Jamia Millia Islamia

New Delhi – 110025

# **TABLE OF CONTENTS**

Abstract.....	6
Chapter 1: Introduction.....	7
1.1 Objective.....	8
1.2 Motivation.....	9
1.3 Overview of Genetic Programming.....	10
1.4 Genetic Programming vs Genetic Algorithms.....	11
Chapter 2: Design Methodology.....	12
2.1 Evolutionary Algorithms Main Idea.....	13
2.2 Program Representation.....	14
2.3 Genetic Operators.....	14
2.4 Meta-Genetic Programming.....	15
Chapter 3: Implementation.....	16
3.1 Program as Trees.....	17
3.2 Creating the Initial Population.....	18
3.3 Testing a Solution.....	19
3.4 Measuring Success.....	19
3.5 Mutating Programs.....	19
3.6 Crossover.....	21
3.7 Building the Environment.....	22
Chapter 4: Result.....	23
4.1 Input Dataset and Primitive Functions.....	24
4.2 Evolved Function.....	25

Technical Details.....	26
Areas of Applications.....	27
Future Research.....	29
References.....	30
Appendix.....	31
A.1 Representing Trees.....	31
A.2 Creating the Initial Population.....	32
A.3 Fitness Test.....	32
A.4 Mutation.....	32
A.5 Crossover.....	32
A.6 Building the Environment.....	33
A.7 Python CGI Script.....	33

## **TABLE OF FIGURES**

1. Genetic Programming Overview.....	11
2. Example Program Tree.....	17
3. Mutation by changing node functions.....	20
4. Mutation by replacing subtree.....	21
5. Crossover operation.....	22
6. Web Interface.....	24
7. Evolved Function.....	25

## **Abstract**

The problem presented in the thesis is of recreating a mathematical function given a dataset. Our approach presents an implementation of symbolic regression which is based on genetic programming (GP). In artificial intelligence, genetic programming is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. It is a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task.

The present approach employs a simple representation for tree-like structures by making use of Read's linear code, leading to more simplicity and better performance when compared with traditional GP implementations. Creation, crossover and mutation of individuals are formalized. An extension allowing for the creation of random coefficients is presented.

This is just a very small sampling of the possibilities of genetic programming—computational power is really the only constraint on the types of problems it can be used to solve.

# **Chapter 1: Introduction**



# **Introduction**

## 1.1 Objective

The objective of our project is to reconstruct the mathematical function from a dataset, also known as Symbolic Regression, using Genetic Programming.

Term "Symbolic Regression" (SR) represents process during which are measured data fitted by suitable mathematical formula like " $x^2 + C$ ", " $\sin(x) + 1/e^x$ " etc. This process is amongst mathematician quite well known and used when some data of unknown process are obtained.

- Example: To evolve an expression whose values match those of the mathematical function  $e^x + \tan(x)$  in the range  $[0, \frac{\pi}{2})$ .

## 1.2 Motivation

For long time "Symbolic Regression" SR was domain only of humans but for a few last decades it is also domain of computers. Idea how to solve various problems by SR by means of evolutionary algorithms (EAs), come from John Koza who used genetic algorithm (GA) in so called genetic programming (GP). Genetic programming is basically symbolic regression which is done by evolutionary algorithms instead of by humans. Ability to solve really hard problems were proved during time many times and GP is today on such level of performance that it is able to synthesise (for example) extremely sophisticated electronic circuits.

It is clear that importance of symbolic regression will increase due to increasing complexity of solved problems in science and industry.

Main attribute of symbolic regression is that it is executed by evolutionary algorithms and whole evolutionary process is working with so called functional set and terminal set. Functional set is a set of all user defined or used functions while terminal set is a set of all constant or variables.

The three different methods for symbolic regression - genetic programming, grammar evolution (GE) and so called analytic programming (AP), which is novelty tool for symbolic regression, based on different principles in regard of GP or GE.

In the 1990s, GP was mainly used to solve relatively simple problems because it is very computationally intensive. Recently GP has produced many novel and outstanding results in areas such as quantum computing, electronic design, game playing, sorting, and searching, due to improvements in GP technology and the exponential growth in CPU power. These results include the replication or development of several post-year-2000 inventions. GP has also been applied to evolvable hardware as well as computer programs.

Developing a theory for GP has been very difficult and so in the 1990s GP was considered a sort of outcast among search techniques.

### 1.3 Overview of Genetic Programming

Genetic programming is a machine-learning technique inspired by the theory of biological evolution. It generally works by starting with a large set of programs (referred to as the *population*), which are either randomly generated or hand-designed and are known to be somewhat good solutions. The programs are then made to compete in some user-defined task. This may be a game in which the programs compete against each other directly, or it may be an individual test to see which program performs better. After the competition, a ranked list of the programs from best to worst can be determined.

Next the best programs are replicated and modified (evolution) in two different ways. The simpler way is *mutation*, in which certain parts of the program are altered very slightly in a random manner in the hope that this will make a good solution even better. The other way to modify a program is through *crossover* (sometimes referred to as *breeding*), which involves taking a portion of one of the best programs and replacing it with a portion of one of the other best programs.

This replication and modification procedure creates many new programs that are based on, but different from, the best programs. At each stage, the quality of the programs is calculated using a *fitness function*. Since the size of the population is kept constant, many of the *worst* programs are eliminated from the population to make room for the new programs. The new population is referred to as “the next generation,” and the whole procedure is then repeated. Because the best programs are being kept and modified, it is expected that with each generation they will get better and better.

New generations are created until a termination condition is reached, which, depending on the problem, can be that:

- The perfect solution has been found.
- A good enough solution has been found.
- The solution has not improved for several generations.
- The number of generations has reached a specified limit.

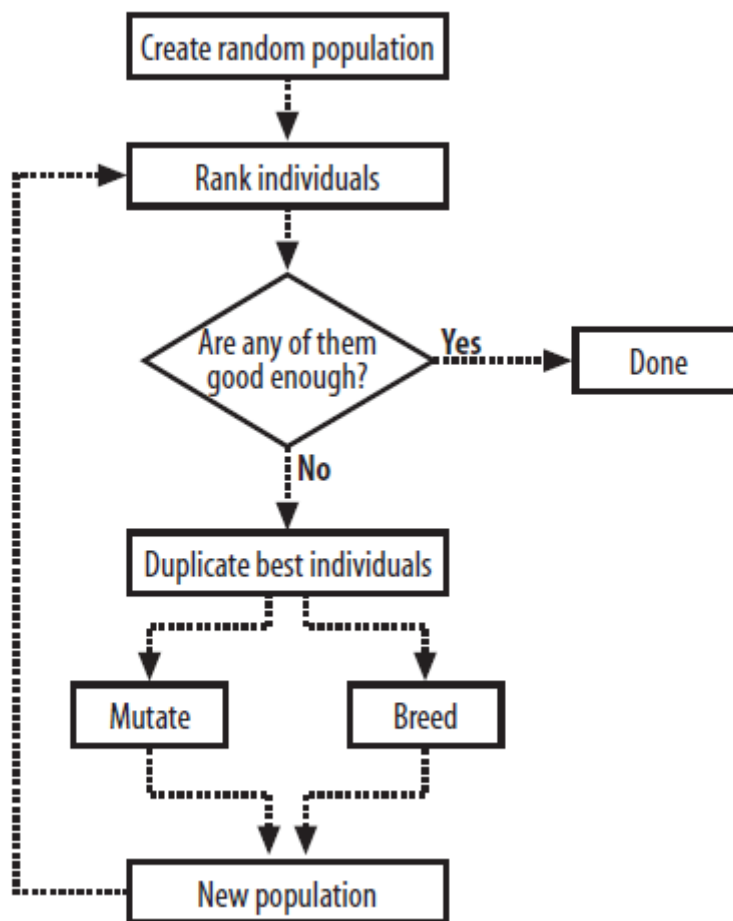
For some problems, such as our project (determining a mathematical function that correctly maps a set of inputs to an output), a perfect solution is possible. For others, such as a board game, there may not be a perfect solution, since the quality of a solution depends on the strategy of the program’s adversary.

An overview of the genetic programming process is shown as a flowchart in Figure 1.

## 1.4 Genetic Programming Versus Genetic Algorithms

Genetic algorithms are an optimization technique that use the idea of evolutionary pressure to choose the best result. With any form of optimization, you have already selected an algorithm or metric and you're simply trying to find the best parameters for it.

Like optimization, genetic programming requires a way to measure how good a solution is; but unlike optimization, the solutions are not just a set of parameters being applied to a given algorithm. Instead, the algorithm itself and all its parameters are designed automatically by means of evolutionary pressure.



*Figure 1: Genetic Programming Overview*

# **Chapter 2: Design**

## **Methodology**

## **Design Methodology**

### 2.1 Evolutionary Algorithms Main Idea

Symbolic regression is in fact based on existence of so called evolutionary algorithms. This class of algorithms is based on Darwinian Theory of evolution and one of its main attributes is that there is no calculated only one solution, but a class of possible solutions at once. This class of possible and acceptable solutions is called "population". Members of this populations are called "individuals" and mathematically said, they represent possible solution, i.e. solution which can be realised in real world application. Main aim of evolutionary algorithms is to find during evolutionary process the best solution of all. Evolutionary algorithms differ among themselves in many points of view like for example individual representation (binary, decimal) or offspring creation (standard crossover, arithmetic operations, vector operations, etc...). They also differ in philosophical background on which they were developed and usually they are named according to this point of view.

Symbolic regression is based on evolutionary algorithms and its main aim is to "synthetize" in an evolutionary way such "program" (mathematical formulas, computer programs, logical expressions, etc...) which will solve user defined problem as well as possible. While domain of EAs is of numerical nature (real, complex, integer, discrete), domain of symbolic regression is of functional nature, i.e. it consist of function set like (sin(), cos(), gamma(), MyFunction(),...) and so called terminal set (t, x, p, ...). From mix of both sets is then synthetized final program, which can be quite complicated in point of view of its structure. In the nowadays there are three methods allowing to do that: genetic programming, grammar evolution and analytic programming. We will consider genetic programming as our evolutionary algorithm for rest of the dissertation.

## 2.2 Program Representation

GP evolves computer programs, traditionally represented in memory as tree structures. Trees can be easily evaluated in a recursive manner. Every tree node has an operator function and every terminal node has an operand, making mathematical expressions easy to evolve and evaluate. Thus traditionally GP favors the use of programming languages that naturally embody tree structures (for example, Lisp; other functional programming languages are also suitable).

Non-tree representations have been suggested and successfully implemented, such as linear genetic programming which suits the more traditional imperative languages [see, for example, Banzhaf *et al.* (1998)]. The commercial GP software Discipulus uses automatic induction of binary machine code ("AIM") to achieve better performance.  $\mu$ GP uses directed multigraphs to generate programs that fully exploit the syntax of a given assembly language.

## 2.3 Genetic Operators

The main operators used in evolutionary algorithms such as GP are crossover and mutation.

- Crossover

Crossover is applied on an individual by simply switching one of its nodes with another node from another individual in the population. With a tree-based representation, replacing a node means replacing the whole branch. This adds greater effectiveness to the crossover operator. The expressions resulting from crossover are very different from their initial parents.

- Mutation

Mutation affects an individual in the population. It can replace a whole node in the selected individual, or it can replace just the node's information. To maintain integrity, operations must be fail-safe or the type of information the node holds must be taken into account. For example,

mutation must be aware of binary operation nodes, or the operator must be able to handle missing values.

## 2.4 Meta-Genetic Programming

Meta-Genetic Programming is the proposed meta learning technique of evolving a genetic programming system using genetic programming itself. It suggests that chromosomes, crossover, and mutation were themselves evolved, therefore like their real life counterparts should be allowed to change on their own rather than being determined by a human programmer. Meta-GP was formally proposed by Jürgen Schmidhuber in 1987. Doug Lenat's Eurisko is an earlier effort that may be the same technique. It is a recursive but terminating algorithm, allowing it to avoid infinite recursion.

Critics of this idea often say this approach is overly broad in scope. However, it might be possible to constrain the fitness criterion onto a general class of results, and so obtain an evolved GP that would more efficiently produce results for sub-classes. This might take the form of a Meta evolved GP for producing human walking algorithms which is then used to evolve human running, jumping, etc. The fitness criterion applied to the Meta GP would simply be one of efficiency.

For general problem classes there may be no way to show that Meta GP will reliably produce results more efficiently than a created algorithm other than exhaustion. The same holds for standard GP and other search algorithms.



# **Chapter 3: Implementation**

## Implementation

### 3.1 Programs As Trees

In order to create programs that can be tested, mutated, and bred, we need a way to represent them. The representation has to lend itself to easy modification and, more importantly, has to be guaranteed to be an actual program. Researchers have come up with a few different ways to represent programs for genetic programming, and the most commonly used is a tree representation.

Most programming languages, when compiled or interpreted, are first turned into a *parse tree*, which is very similar to what we'll be working with here. (The programming language Lisp and its variants are essentially ways of entering a parse tree directly.) An example of a parse tree is shown in Figure 2.

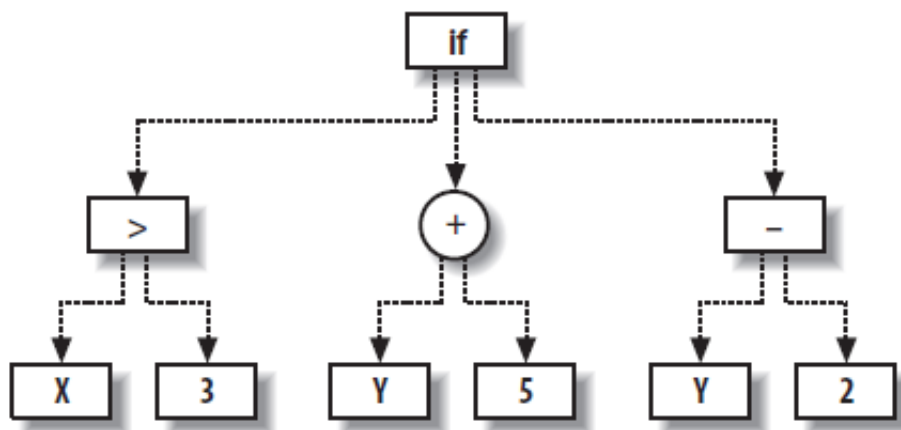


Figure 2. Example program tree

Each node represents either an operation on its child nodes or an endpoint, such as a parameter with a constant value. For example, the circular node is a sum operation on its two branches, in this case, the values Y and 5. Once this point is evaluated, it is given to the node above it, which in turn applies its own operation to its branches.

We also observe that one of the nodes has the operation “if,” which specifies that if its leftmost branch evaluates to true, return its center branch; if it doesn’t, return its rightmost branch.

At first, it might appear that these trees can only be used to build very simple functions. There are two things to consider here—first, the nodes that compose the tree can potentially be very complex functions, such as distance measures or Gaussians. The second thing is that trees can be made recursive by referring to nodes higher up in the tree. Creating trees like this allows for loops and other more complicated control structures.

Refer to Appendix A.1 “Representing Trees” for the python implementation.

## 3.2 Creating the Initial Population

Although it’s possible to hand-create programs for genetic programming, most of the time the initial population consists of a set of random programs. This makes the process easier to start, since it’s not necessary to design several programs that almost solve a problem. It also creates much more *diversity* in the initial population—a set of programs designed by a single programmer to solve a problem are likely to be very similar, and although they may give answers that are almost correct, the ideal solution make look quite different. Thus, diversity is very important.

Refer to Appendix A.2 “Creating the Initial Population” for the python implementation.

### 3.3 Testing a Solution

Theoretically, to find a solution to our problem, we could just generate random programs until one is correct. Obviously, this would be ridiculously impractical because there are infinite possible programs and it's highly unlikely that we would stumble across a correct one in any reasonable time frame.

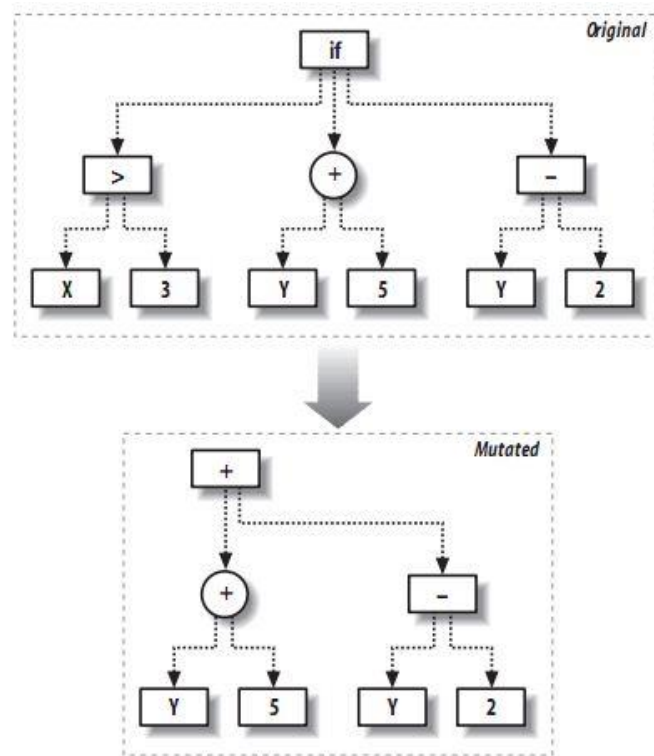
### 3.4 Measuring Success

As with optimization, it's necessary to come up with a way to measure how good a solution is. An easy way to test a program is to see how close it gets to the correct answers for the dataset. We can make a function that checks every row in the dataset, calculating the output from the function and comparing it to the real result. It adds up all the differences, giving lower values for better programs—a return value of 0 indicates that the program got every result correct. Refer to Appendix A.3 “Fitness Test” for the python implementation.

### 3.5 Mutating Programs

After the best programs are chosen, they are replicated and modified for the next generation. Mutation takes a single program and alters it slightly. The tree programs can be altered in a number of ways—by changing the function on a node or by altering its branches. A function that changes the number of required child nodes either deletes or adds new branches, as shown in Figure 3. The other way to mutate is by replacing a subtree with an entirely new one, as shown in Figure 4.

We must remember that the mutations are random, and they aren't necessarily directed toward improving the solution. The hope is simply that some will improve the result. These changes will be used to continue, and over several generations the best solution will eventually be found. Refer to Appendix A.4 “Mutation” for the python implementation.



*Figure 3. Mutation by changing node functions*

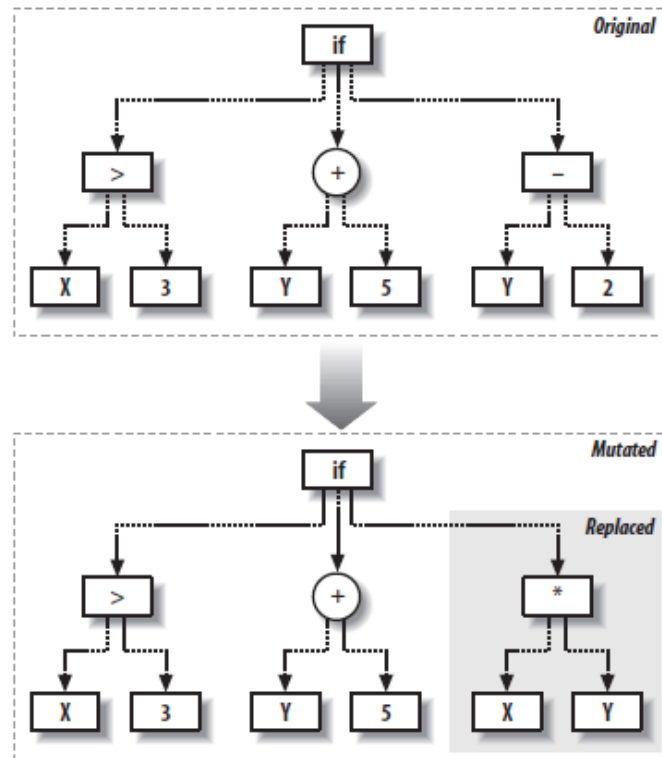


Figure 4. Mutation by replacing subtrees

### 3.6 Crossover

The other type of program modification is crossover or breeding. This involves taking two successful programs and combining them to create a new program, usually by replacing a branch from one with a branch from another. Figure 5 shows an example of how this works.

Refer to Appendix A.5 “Crossover” for the python implementation.

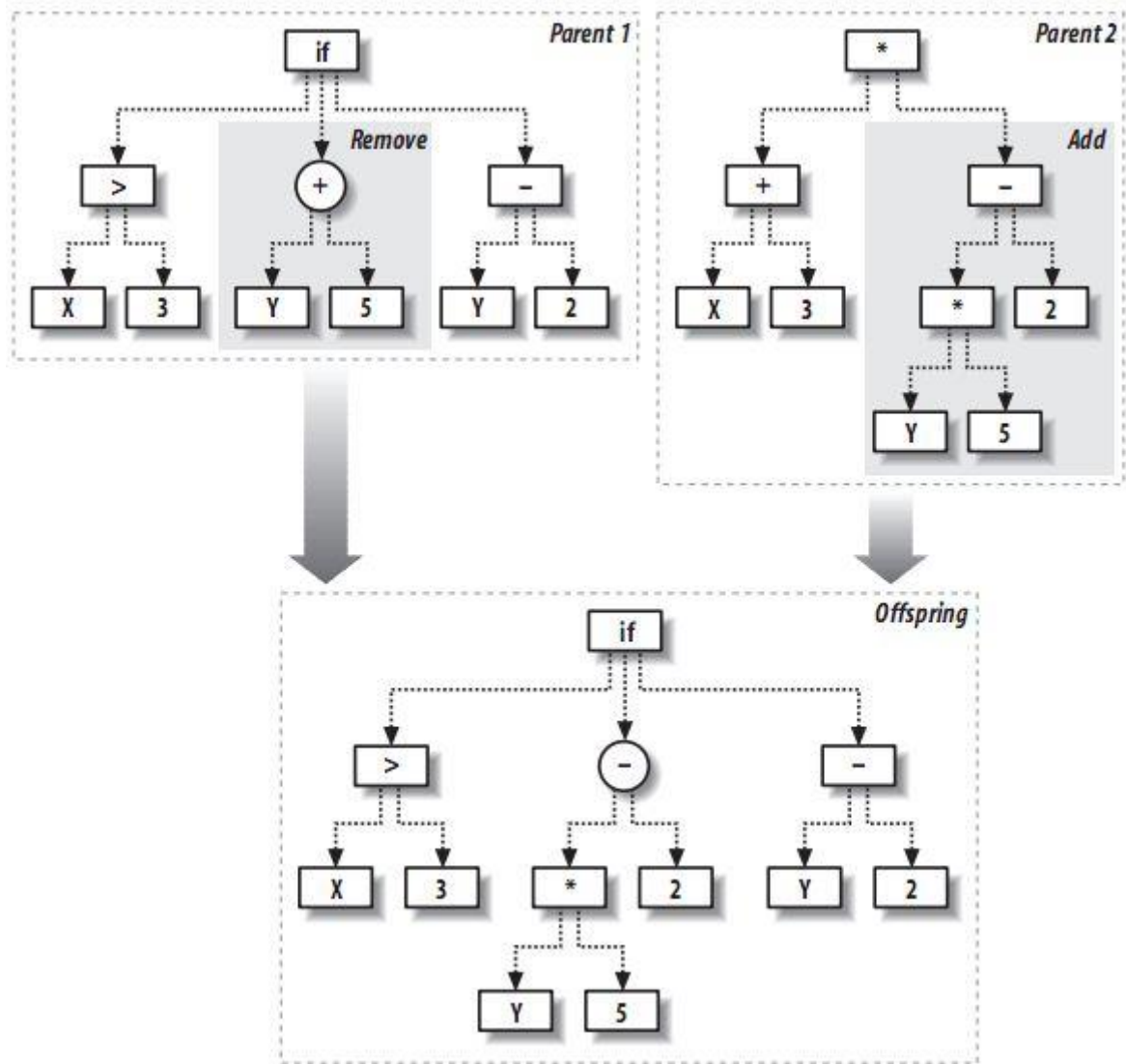


Figure 5. Crossover operation

### 3.7 Building the Environment

Armed with a measure of success and two methods of modifying the best programs, we are now ready to setup a competitive environment in which programs can evolve. The steps are shown in the flow chart in Figure 1. Essentially, we create a set of random programs and select the best ones for replication and modification, repeating this process until some stopping criteria is reached.

Refer to Appendix A.6 “Building the Environment” for the python implementation

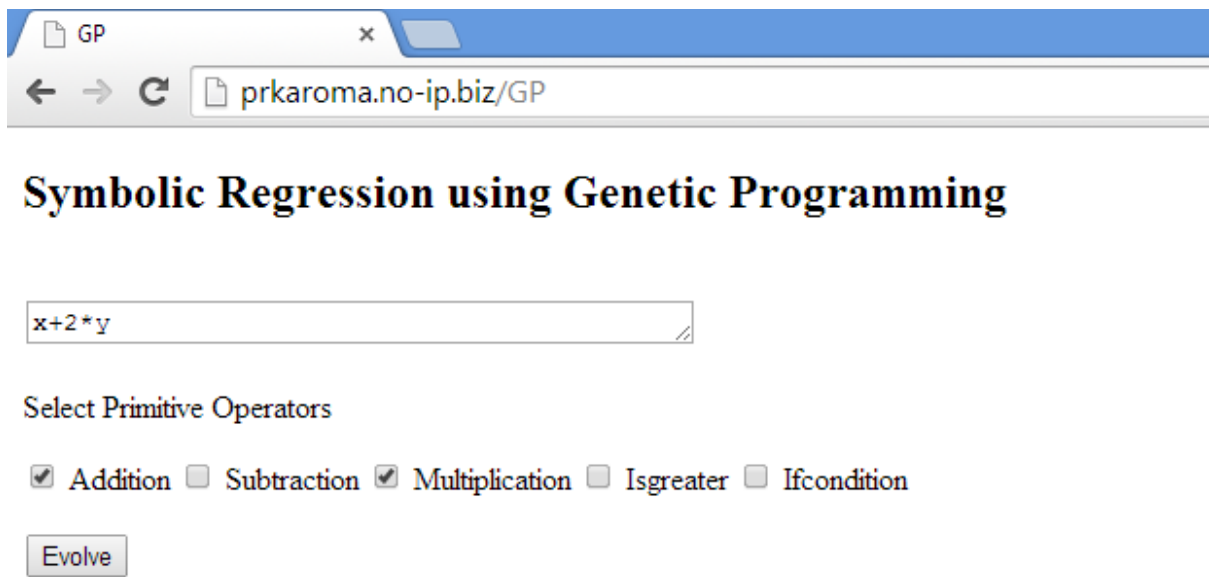
# **Chapter 4: Result**



## Result

### 4.1 Input Dataset and Primitive Operators

Input dataset is created through a hidden function provided in the applications as shown below.



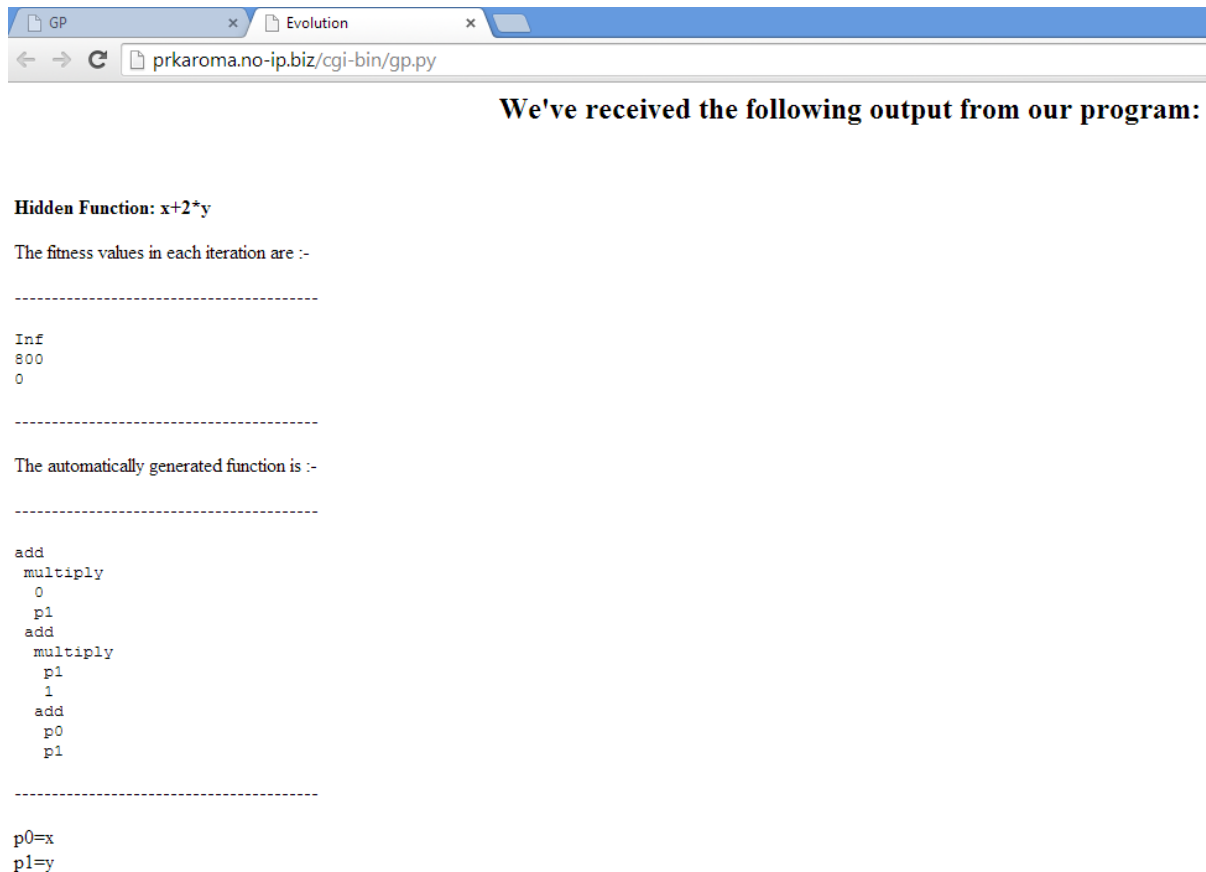
*Figure 6. Web Interface*

Primitive Functions fed into application are:

1. Addition
2. Subtraction
3. Multiplication
4. Greater than
5. If Condition

## 4.2 Evolved Function

The evolved function returned by the application for the corresponding input function i.e.  $x + 2 * y$  is shown below.



```
GP Evolution
prkaroma.no-ip.biz/cgi-bin/gp.py

Hidden Function: x+2*y

The fitness values in each iteration are :-

-----

Inf
800
0

-----

The automatically generated function is :-

-----

add
multiply
  0
  p1
add
multiply
  p1
  1
add
  p0
  p1

-----

p0=x
p1=y
```

**We've received the following output from our program:**

*Figure 7. Evolved Function*

It's very likely that the solution generated will seem more complicated than it has to be. However, a little algebra shows us that these functions are actually the same—remember that  $p0$  is  $x$  and  $p1$  is  $y$ .

The function returned by the program is  $(0 * p1) + ((p1 * 1) + (p0 + p1)) = p0 + 2 * p1$

## **Technical Details**

<b>Programming Language</b>	Python
<b>Web Language</b>	Python CGI Scripting
<b>Number of variables</b>	2
<b>Number of primitive functions</b>	5
<b>Range of constants</b>	0 – 10
<b>Mutation Probability</b>	0.2
<b>Crossover Probability</b>	0.1

## **Areas of Applications**

The primary application of our project is in the field of Statistics. Given a statistical dataset, it helps to find a mathematical function (if one exists) that relates the various variables of the dataset. It becomes increasingly important to do so if the dataset is huge, for we can simply use the mathematical function to calculate the dataset values, instead of storing the huge dataset and performing costly search and retrieval operations on it. Thus, it helps reduce the computational and memory costs. Our method using Genetic programming is better than regression analysis (which would require guessing the structure of the formula first) and K-nearest neighbors method (which would require keeping all the data).

There are numerous other applications of genetic programming.

- The genetic programming technique has been applied in designing antennas for NASA.
- It has been used in photonic crystals, optics, quantum computing systems, and other scientific inventions.
- It has also been used to develop programs for playing many games, such as chess and backgammon. In 1998, researchers from Carnegie Mellon University entered a robot team that was programmed entirely using genetic programming into the RoboCup soccer contest, and placed in the middle of the pack.
- “Black Art Problems,” such as the automated synthesis of analog electrical circuits, controllers, antennas, networks of chemical reactions, optical systems, and other areas of design,
- “Programming The Unprogrammable” (PTU) involving the automatic creation of computer programs for unconventional computing devices such as cellular automata,

multi-agent systems, parallel programming systems, field-programmable gate arrays, field-programmable analog arrays, ant colonies, swarm intelligence, distributed systems, and the like.

- Commercially Useful New Inventions (CUNI) involving the use of genetic programming as an automated "invention machine" for creating commercially usable new inventions.

## **Future Research**

We have used a very small set of functions to construct the programs so far. For more complicated problems, it's necessary to greatly increase the number of functions available to build a tree.

Here are some possible functions to add:

- Statistical distributions, such as a Gaussian
- Distance metrics, like Euclidean and Tanimoto distances
- A three-parameter function that returns 1 if the first parameter is between the second and third
- A three-parameter function that returns 1 if the difference between the first two parameters is less than the third

This project can only handle integer and float values. But the basic framework can be altered to handle more complex data types such as

- Strings: These would have operations like concatenate, split, indexing, and substrings.
- Lists: These would have operations similar to strings.
- Dictionaries: These would include operations like replacement and addition.
- Objects: Any custom object could be used as an input to a tree, with the functions on the nodes being method calls to the object.

## **References**

- [1] Banzhaf, W., Nordin, P., Keller, R.E., and Francone, F.D. (1998), *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*, Morgan Kaufmann
- [2] Koza, J.R. (1990), *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*, Stanford University Computer Science Department technical report STAN-CS-90-1314.
- [3] Segaran, Toby (2007), *Programming Collective Intelligence: Building Smart Web 2.0 Applications*.
- [4] Langdon, W. B., Poli, R. (2002), *Foundations of Genetic Programming*, Springer-Verlag ISBN 3-540-42451-2
- [5] Various websites like <http://www.genetic-programming.com>, Wikipedia, [ieeexplore](http://ieeexplore.org), <http://mitpress.mit.edu>, etc.

## Appendix

### A.1 Representing Trees

```
class fwrapper:
    def __init__(self,function,childcount,name):
        self.function=function
        self.childcount=childcount
        self.name=name

class node:
    def __init__(self,fw,children):
        self.function=fw.function
        self.name=fw.name
        self.children=children
    def evaluate(self,inp):
        results=[n.evaluate(inp) for n in self.children]
        return self.function(results)
    def display(self,indent=0):
        print (' '*indent)+self.name
        for c in self.children:
            c.display(indent+1)

class paramnode:
    def __init__(self,idx):
        self.idx=idx
    def evaluate(self,inp):
        return inp[self.idx]
    def display(self,indent=0):
        print '%sp%d' % (' '*indent,self.idx)

class constnode:
    def __init__(self,v):
        self.v=v
    def evaluate(self,inp):
        return self.v
    def display(self,indent=0):
        print '%s%d' % (' '*indent,self.v)
```



## A.2 Create Initial Population

```
def makerandomtree(pc,maxdepth=4,fpr=0.5,ppr=0.6):
    if random()<fpr and maxdepth>0:
        f=choice(flist)
        children=[makerandomtree(pc,maxdepth-1,fpr,ppr)
                  for _ in range(f.childcount)]
        return node(f,children)
    elif random()<ppr:
        return paramnode(randint(0,pc-1))
    else:
        return constnode(randint(0,10))
```

## A.3 Fitness Test

```
def scorefunction(tree,s):
    dif=0
    for data in s:
        v=tree.evaluate([data[0],data[1]])
        dif+=abs(v-data[2])
    return dif
```

## A.4 Mutation

```
def mutate(t,pc,probchange=0.1):
    if random()<probchange:
        return makerandomtree(pc)
    else:
        result=deepcopy(t)
        if isinstance(t,node):
            result.children=[mutate(c,pc,probchange) for c in t.children]
        return result
```

## A.5 Crossover

```
def crossover(t1,t2,probswap=0.7,top=1):
    if random()<probswap and not top:
        return deepcopy(t2)
    else:
        result=deepcopy(t1)
        if isinstance(t1,node) and isinstance(t2,node):
            result.children=[crossover(c,choice(t2.children),probswap,0)
                             for c in t1.children]
        return result
```

## A.6 Building the Environment

```
def evolve(pc,popsize,rankfunction,maxgen=500,
          mutationrate=0.1,breedingrate=0.4,pexp=0.7,pnew=0.05):
    # Returns a random number, tending towards lower numbers. The lower pexp
    # is, more lower numbers you will get
    def selectindex():
        return int(log(random())/log(pexp))
    # Create a random initial population
    population=[makerandomtree(pc) for _ in range(popsize)]
    for _ in range(maxgen):
        scores=rankfunction(population)
        print scores[0][0]
        if scores[0][0]==0: break
        # The two best always make it
        newpop=[scores[0][1],scores[1][1]]
        # Build the next generation
        while len(newpop)<popsize:
            if random()>pnew:
                newpop.append(mutate(
                    crossover(scores[selectindex()][1],
                             scores[selectindex()][1],
                             probswap=breedingrate),
                    pc,probchange=mutationrate))
            else:
                # Add a random node to mix things up
                newpop.append(makerandomtree(pc))
        population=newpop
        scores[0][1].display()
    return scores[0][1]
```

## A.7 CGI Script

```
#!/usr/bin/python

# Import modules for CGI handling
import cgi, cgitb

from random import random,randint,choice
from copy import deepcopy
from math import log

# Create instance of FieldStorage
form = cgi.FieldStorage()
```

```

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Evolution</title>"
print "</head>"
print "<body>"
print "<h2><center>We've received the following output from our program: </center></h2><br/>"
print "<p><b>Hidden Function: %s</b></p>" % text_content
print "<p>The fitness values in each iteration are :-</p>"
print "<p>-----</p>"
print "<pre>%s</pre>" % '\n'.join(s)
print "<p>-----</p>"
print "<p>The automatically generated function is :-</p>"
print "<p>-----</p>"
print "<pre>%s</pre>" % '\n'.join(out)
print "<p>-----</p>"
print "<p>p0=x<br>p1=y</p>"
print "</body>"
print "</html>"

```