University of St. Thomas

Graduate Programs in Software

SEIS 610: Software Engineering Spring 2014

Genetic Programming

Final Report

5/19/2014

Maryam Alruwaythi
Jed Brundidge
Tabinda Hasan
Kelly Heitz
Dawn Young

Table of Contents

Preface

All of the documents referenced here, and historical documentation of the life of this project from beginning to end, are accessible as Google docs in *GP group project* folder. Anyone with this link has access: https://drive.google.com/folderview?id=0BxP-i4iTYfT5LWtKcDFOU1VvbDQ&usp=sharing

Our final genetic programming system was a modification of package called tiny_gp we found online, with guidance from a source called "A Field Guide to Genetic Programming" by Riccardo Poli, William B. Langdon, and Nicholas F. McPhee, copyright March 2008.

Link to the original tiny_gp: http://cswww.essex.ac.uk/staff/rpoli/TinyGP/

Project Summary

Problem Definition

Create a genetic programming application that will automatically evolve based on experiences the system "learns" by evaluating fitness of potential solutions against an established set of training data. The evolution will occur through the use of crossover and mutation methodologies applied probabilistically to the initial and subsequent generations of potential solutions, with the most "fit" solutions having the highest probability of representation in subsequent generations. Specifically, the program will randomly generate an initial set of functions and then repeatedly measure, modify, and regenerate functions until one that is equivalent (or very close) to the target function f(x) = (2x2+4)/3 is achieved within a 15-minute time frame.

The project shall be cooperatively managed, created, and tested by the team, presented on May 12th, and submitted with all supporting documentation and post-project analysis on May 19th, 2014.

Project Requirements

The following requirements include mark-ups to demonstrate how our view the project changed over time. All but one of the requirement changes were based on decisions our group made as we acquired more knowledge and experience along the way. The single requirement change from the customer, Professor Lai, was the target function in requirement 8c.

1. Input Expectations
   a. Training data file (target function implied)
      i. List (0, 1/2), (1, 0), (-1, 0), etc.
   b. Function set
      i. +, -, *, /
   c. Terminal set
      i. 0, 1, 2, 3, **4, 5, 6, 7, 8, 9**, x
   d. Population size of each generation

~~i. 20~~ **100,000**

2. Binary Tree requirements
    a. Each Operator node (functional node) must have 2 child nodes
    b. Each Operand (terminal node) must have 0 child nodes
3. Generating the initial population, generation 1, G1
    a. Use ~~"Full Method"~~ **"Grow Method"** to randomly generate ~~20~~ **100,000** functions
    b. Operators will be randomly selected from the Function Set
    c. Operands will be randomly selected from the Terminal Set.
    d. Depth of the initial binary trees will be 4 **3**
        i. Each node in level 1, 2, ~~and 3~~ must have 2 child nodes.
        ii. Each node in level 4 **3** must have 0 child nodes
4. Fitness
    a. Calculate the fitness of each function (f1 through f20) in the population in each generation (G1, G2, etc.).
    b. For each function in the population (f1 through f20), calculate its fitness as follows:
        i. Apply the function to each x in the Training Data Set (x1 through x20) and retain the result as G1f1x1, G1f1x2, etc
        ii. Subtract the function's result for each x from that x's result in the training set to find the difference, d:
            1. For example, referring to the training set function as f0 (the target function):
               G1f1x1 – f0x1 = dG1f1x1 (this is the difference d for x1 using function 1 in generation 1)
               G1f1x2 – f0x2 = dG1f1x2
               G1f1x3 – f0x3 = dG1f1x3
            2. Note that f0x1 through f0x20 will always be the same in every calculation in every generation because f0 represents the target function applied to each x in the Training Data Set.
        iii. Take the absolute value of every difference
        iv. Fitness of a function is the sum of the absolute values of the differences.
        1. For example, the fitness of function 1 in generation 1 is calculated as follows:
           FG1f1 = |dG1f1x1| + |dG1f1x2| + |dG1f1x3| …. |dG1f1x20|
    ~~c. Rank the 20 functions by their fitness:~~
        ~~i. Rank 1 will have the lowest fitness value~~
        ~~ii. Rank 20 will have the highest fitness value~~
    ~~d. For each generation, sum the fitness values of the top 5 ranked functions and store the sum as T5G1, T5G2, etc. (This value will be used later to compare the fitness of generations over time.)~~
5. Varying the population from generation to generation
    a. Start with ~~20~~ **100,000** functions from the preceding generation ~~(they should be ranked in order of their fitness to the training set).~~
    ~~b. Eliminate 10% of the lowest ranked functions, i.e. eliminate functions ranked 19 and 20.~~
    ~~c. Reproduce (clone) 10% of the highest ranked functions, i.e. reproduce functions ranked 1 and 2.~~
        ~~i. 2 of the 20 functions will be the most fit function from the prior generation~~
        ~~ii. 2 of the 20 will be the 2nd most fit function from the prior generation~~
        iii. ~~The 16 other functions will be the remaining functions from the prior generation (excluding the 2 least fit functions).~~
    d. Use tournament selection.

        i. **Tournament size = 4**
        ii. **Negative tournaments are used for the selection of the individuals to be replaced at each steady-state-GP iteration.**

    e. Crossover:
        i. ~~Randomly select 2 functions from list of 20 functions~~
        ii. ~~From the higher ranked function, randomly select a node that will be crossed over.~~
        iii. ~~In the lower ranked function, randomly select a node that will be replaced.~~
        iv. ~~Replace the node in the lower ranked function with the node from the higher ranked function.~~
        v. ~~Repeat 10 times~~
        vi. ~~The fitness rank of each function will remain static in spite of crossover changes occurring in the current generation.~~
        vii. **Subtree crossover is used. The selection of crossover points is uniform, so every node is chosen equally likely.**

    e. Mutation

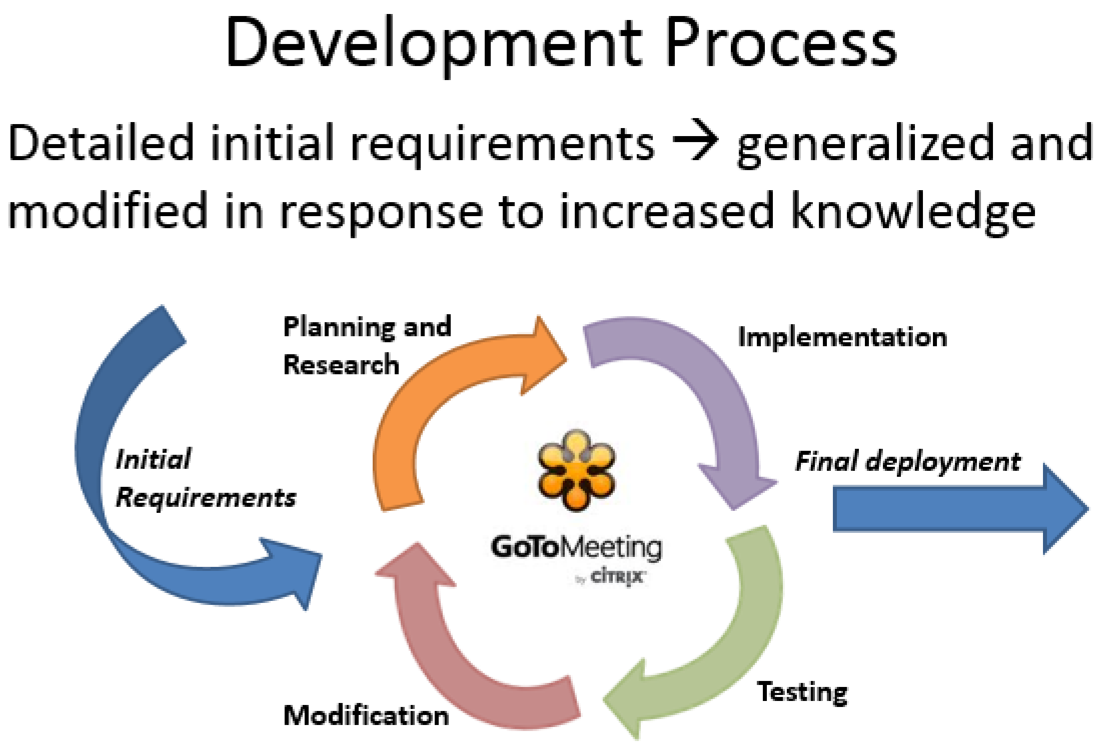
        i. ~~Apply mutation to the new set of 20 functions with probability of 0.01 (assuming 1 function in 5 generations)~~
        ii. ~~Set an indicator to mark "yes" or "no" to each function, indicating whether it was selected for mutation or not.~~
        iii. Randomly select any node, a functional node or a terminal node
            1. ~~Whether the selected node is functional or terminal, initialize a new sub-tree with a depth of 3 using "Full" method.~~
            2. **Apply point mutation to the selected node: i.e., if a functional node is selected randomly replace it with a different functional node, and vice versa.**

6. ~~Evaluate the general population after each 20th generation~~
    a. ~~If T5G20 > T5G10, eliminate lowest 50% fit functions~~
        i. ~~Use strategy we used to generate the initial population to add 10 new functions to the population set~~
        ii. ~~At 20th, 40th, 60th generations, etc. the sum of the current top 5 fitness functions should be less than the sum of the top 5 fitness factors from 10 generations prior.~~
7. Output Expectations
    a. ~~During development phase,~~
        i. ~~print after each generation the population of 20 functions that have been generated~~
        ii. ~~Rank them by fitness~~
        iii. ~~Which functions were eliminated?~~
        iv. ~~Which functions were reproduced?~~
        v. ~~Which functions were mutated?~~
    b. After programming is finalized, **print for each generation:**
        i. Print best function
            1. Which generation produced it
            2. Its fitness
            3. The average fitness of the ~~final~~ generation
            4. **Average function length of the generation**

        5. Length of time the program ran
        ~~6. A fitness graph comparing the best function against the target function~~

8. Acceptance/Termination criteria
    a. Terminate program when fitness of ~~any function in the population set = 0~~ **the best function is <= .0009**, or
    b. After 15 minute limit
        i. Select most fit function
    c. Function should be equivalent to: ~~$f(x) = (x^2 – 1)/2$~~ **$f(x) = (2X^2 +4)/3$**
    d. Program will run without errors or crashing.

Development Process



Development Process

Detailed initial requirements → generalized and modified in response to increased knowledge

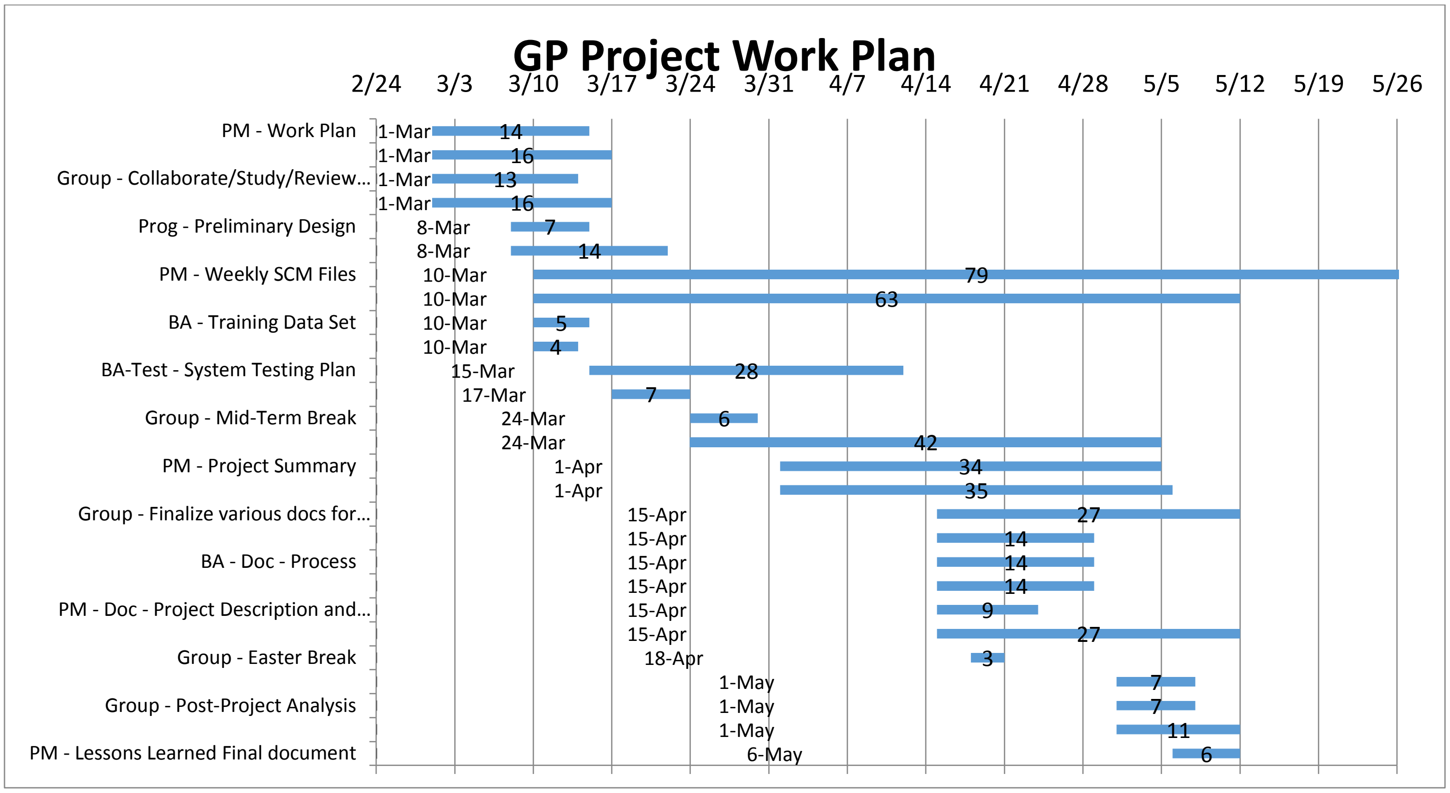
Project Implementation

Team Roles

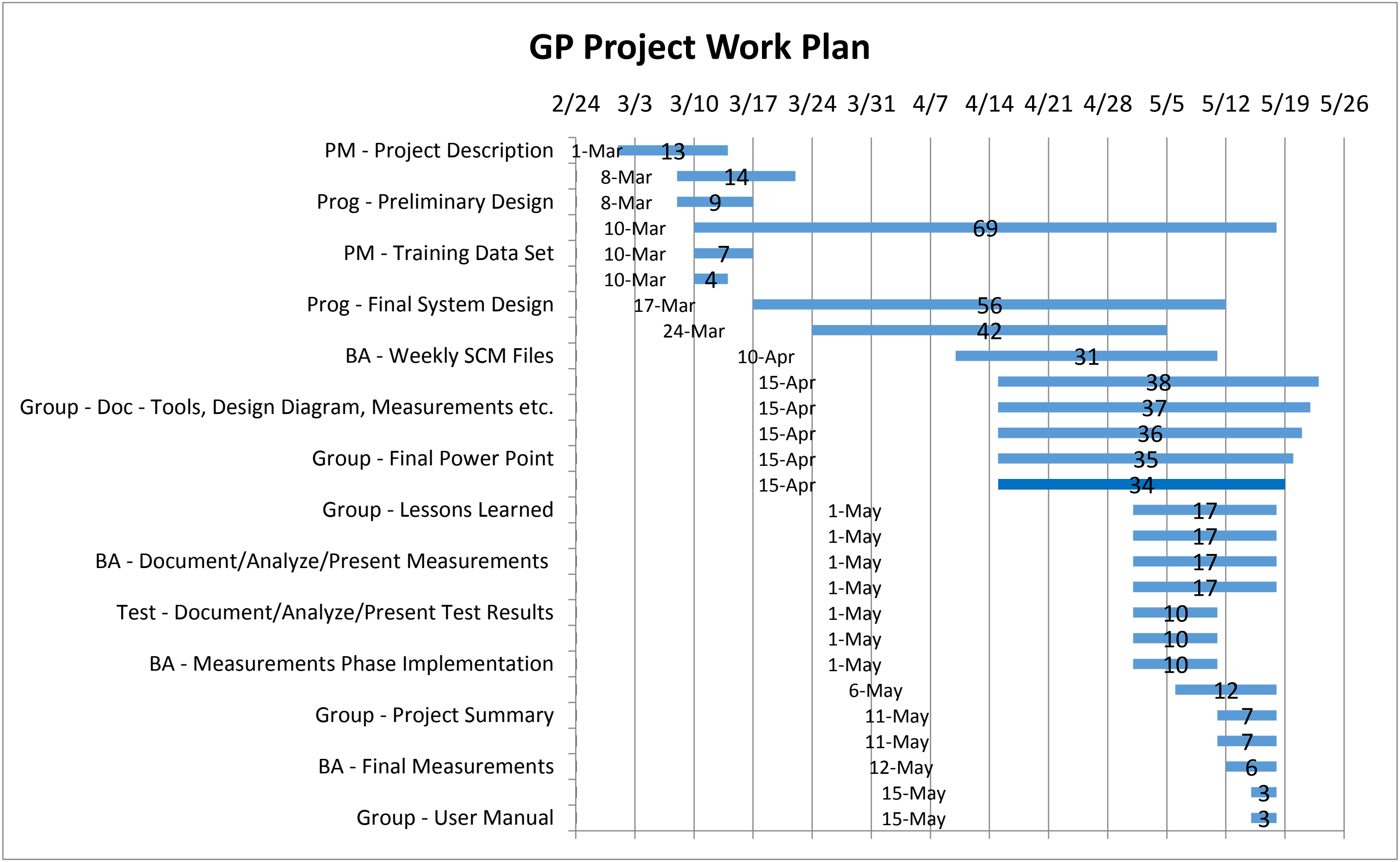
Member	Starting Role	Mid-term Role	End of Project Role
Maryam	Analyze/Develop Reqmts Test Code	Analyze/Develop Reqmts Analyze Packages	Research Package Code Measure Code
Jed	Develop Code	Analyze/Develop Reqmts Develop Code	Analyze Package Code Develop Test Plan Test Code
Tabinda	Analyze/Develop Reqmts Co-Manage Project	Analyze/Develop Reqmts Analyze Packages	Analyze Package Code Measure Code
Kelly	Develop Code	Analyze/Develop Reqmts Create UML diagram Develop Code	Modify Package Code Modify Requirements Modify UML diagram Black box testing
Dawn	Analyze/Develop Reqmts Co-Manage Project	Analyze/Develop Reqmts Develop Code Admin Meetings	Modify Package Code Compile Presentation Compile Final Report Manage Project

Timeline

Initial Timeline:
We started the project with the following timeline on when we expected to complete tasks assigned to the group members. Initially we were under the impression that we knew exactly how much time we were expected to (and the amount of time we should/would spend on the assigned tasks). During the course of learning more about the project and the expectations from the Genetic Programming Application we were developing, many of the aforementioned tasks changed, as did the amount we spent on them as a group. We had allocated a large amount of time for meeting and dissecting the requirements & design, and then later for evaluating the program we had started to build.

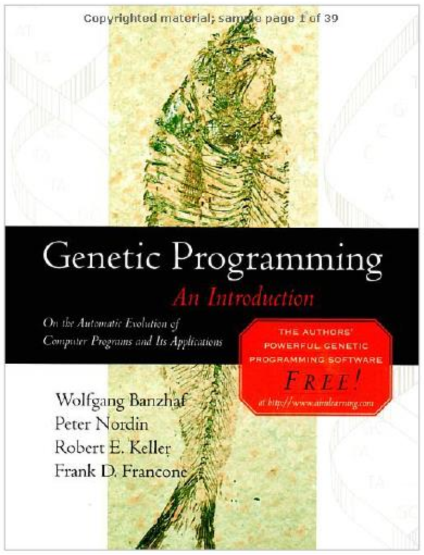


Actual Timeline:
As expected we spent a large portion of our time in finalizing the requirements, design, then developing the program and pulling together all of the documentation and visual materials together. We did eventually decide to go with a package and a lot of the final couple of weeks were spent on getting the code of this new package to return the results we expected from our GP Application, in addition to measuring, testing and then accumulating the visual materials and documentation to support our finished product (i.e. our GP Application). It would have been ideal to spend a lot more time on testing and measuring this new code but with very few requirements and no team experience with packages, the search for an appropriate package for modification took longer than anticipated.



Tools and Resources

Google Drive


Copyrighted material; sample page 1 of 39

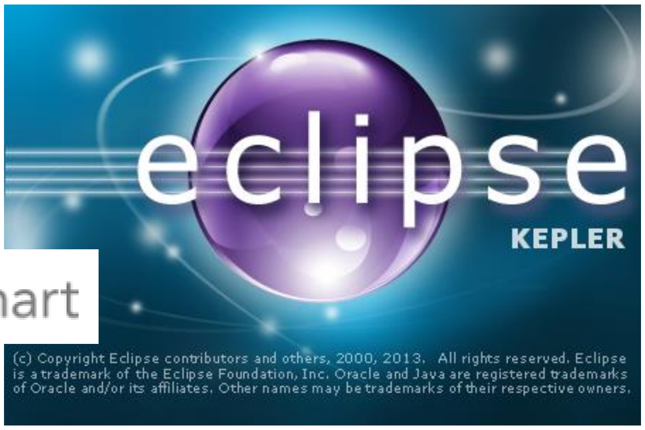
Genetic Programming

An Introduction

On the Automatic Evolution of Computer Programs and Its Applications

THE AUTHORS' POWERFUL GENETIC PROGRAMMING SOFTWARE FREE! at http://www.aimlearning.com

Wolfgang Banzhaf
Peter Nordin
Robert E. Keller
Frank D. Francone



(c) Copyright Eclipse contributors and others, 2000, 2013. All rights reserved. Eclipse is a trademark of the Eclipse Foundation, Inc. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Lucidchart

CodePro AnalytiX™

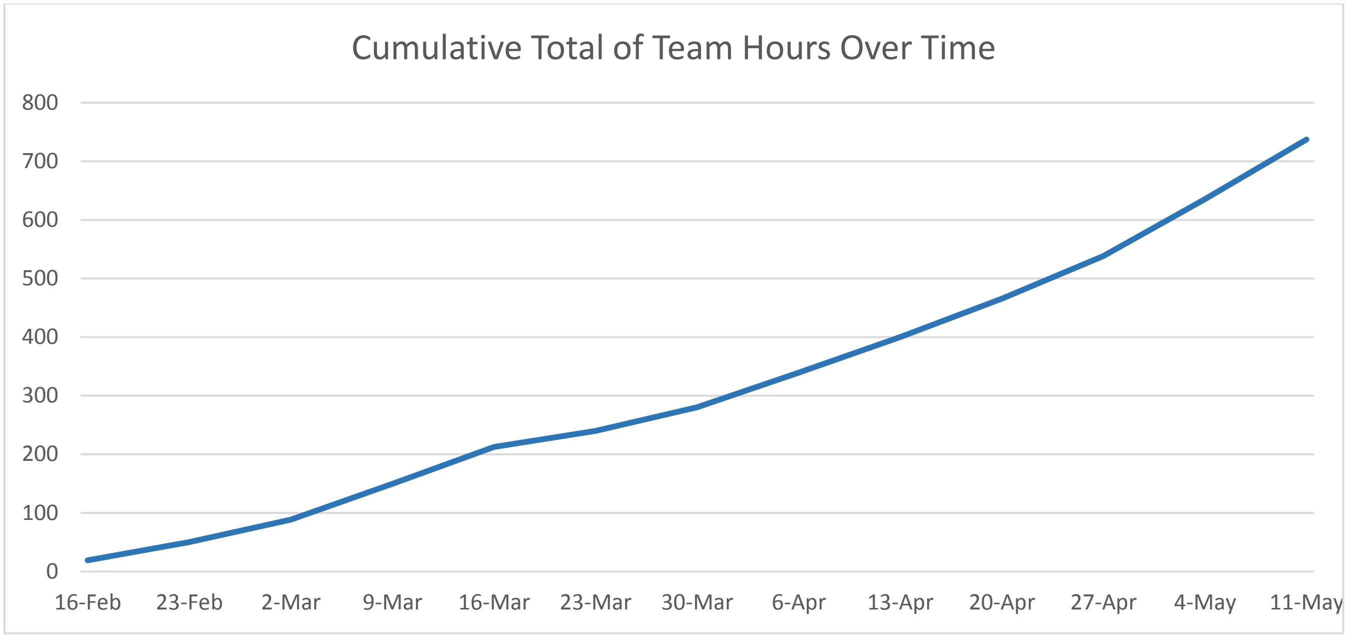
GoToMeeting

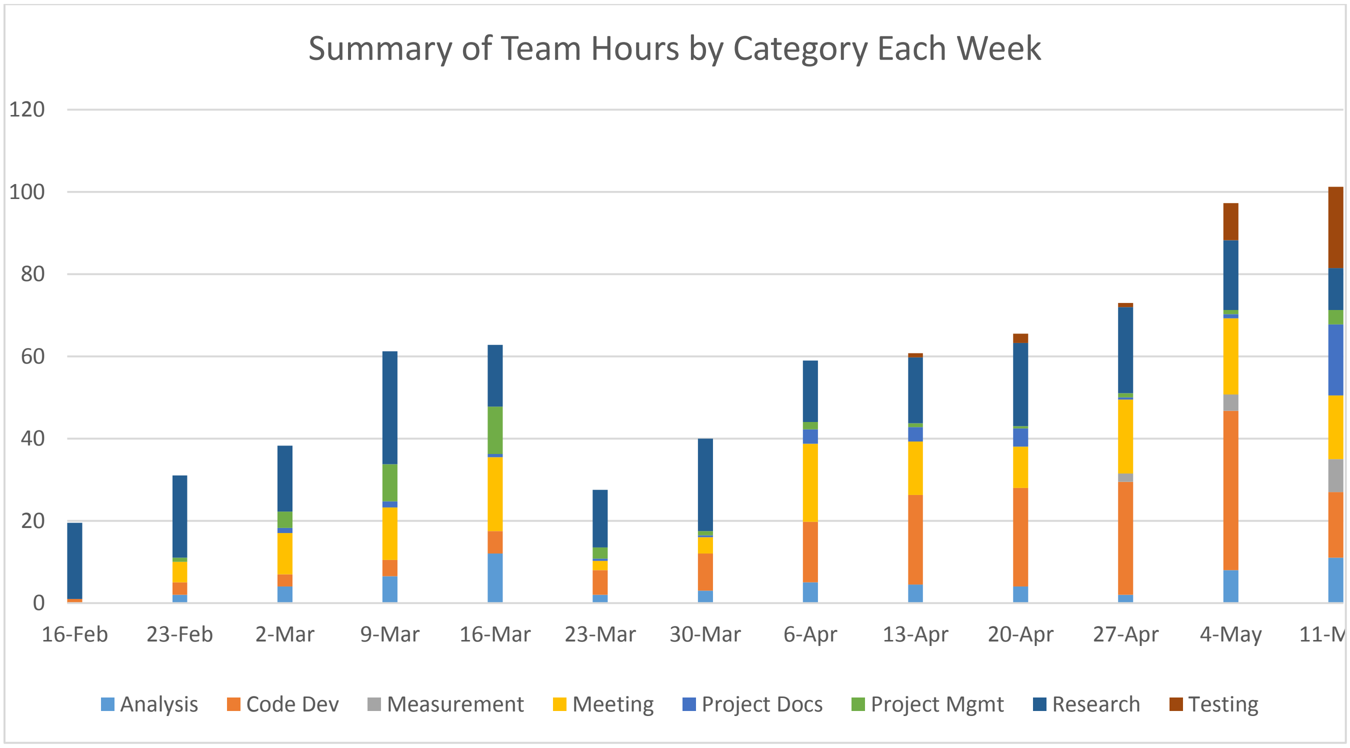
stackoverflow

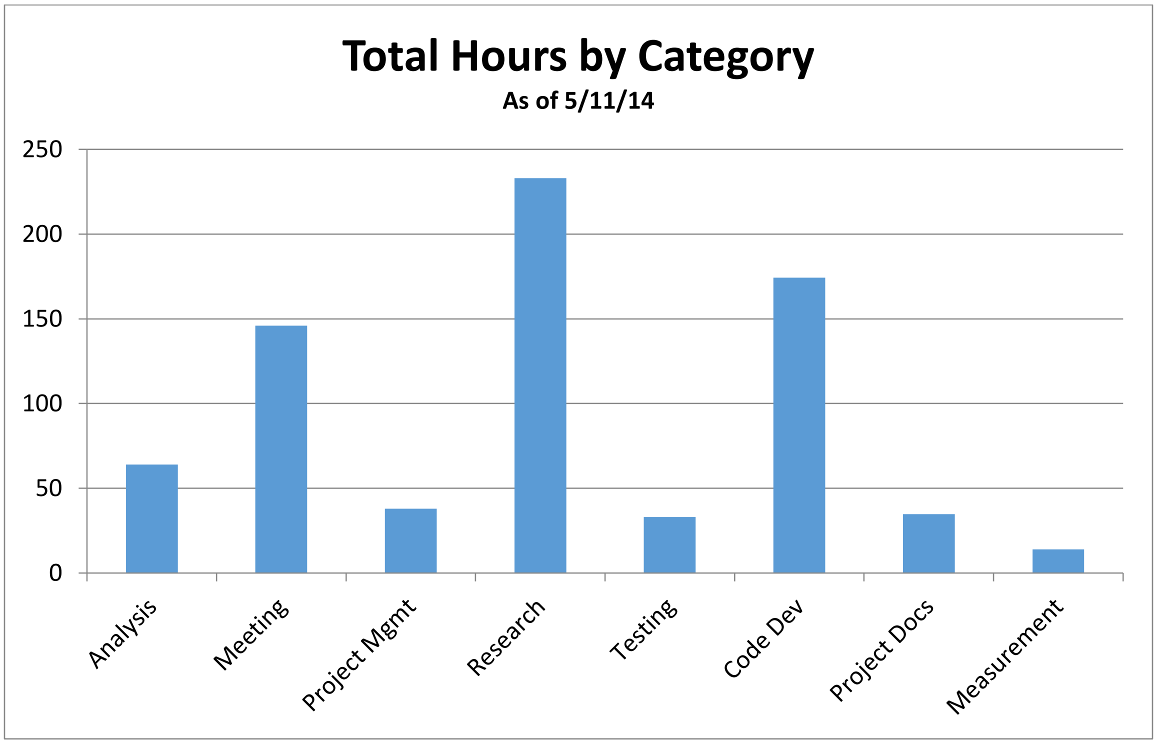
A FIELD GUIDE TO GENETIC PROGRAMMING

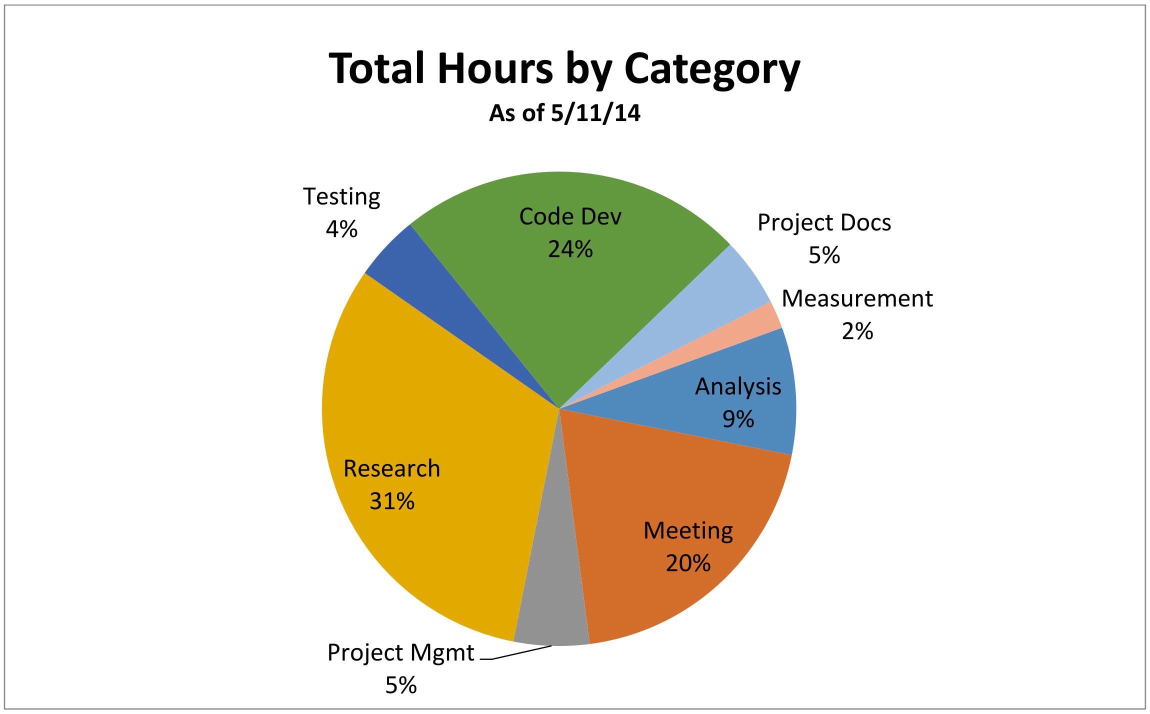
WEB SITE IN SUPPORT OF A FREELY DOWNLOADABLE, CREATIVE COMMONS LICENSED INTRODUCTION TO GENETIC PROGRAMMING

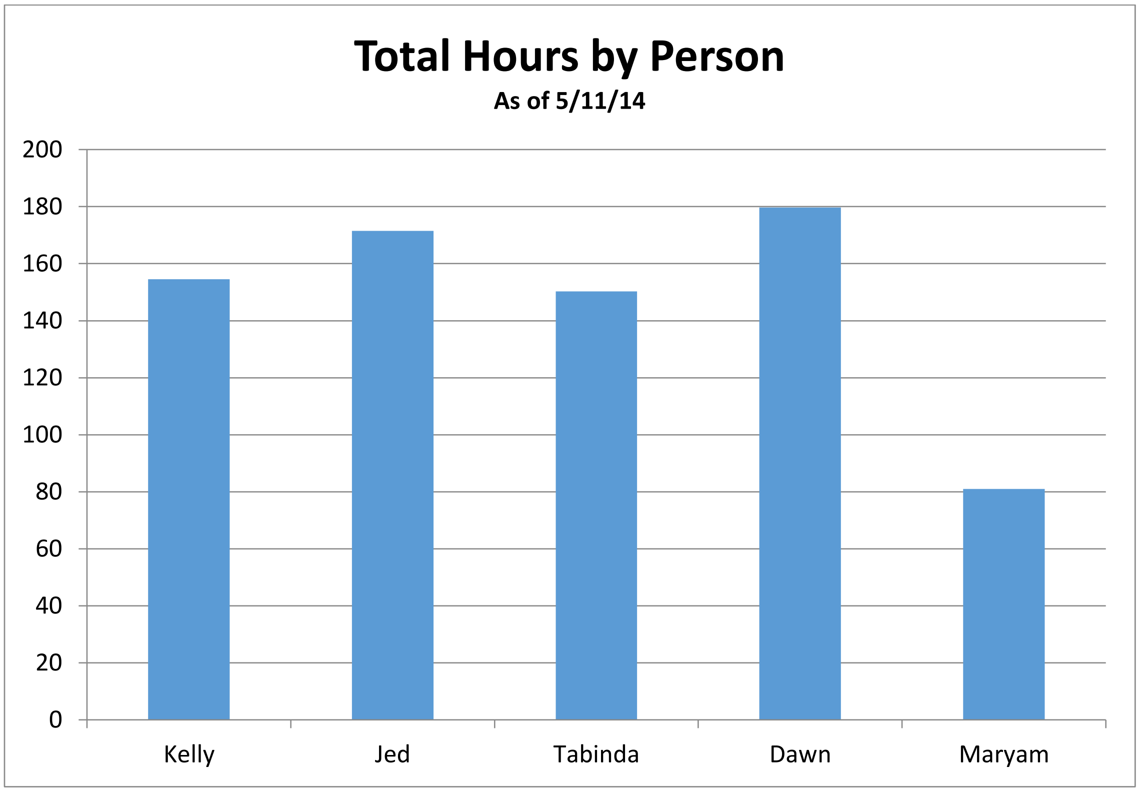
Team Activity Measurements

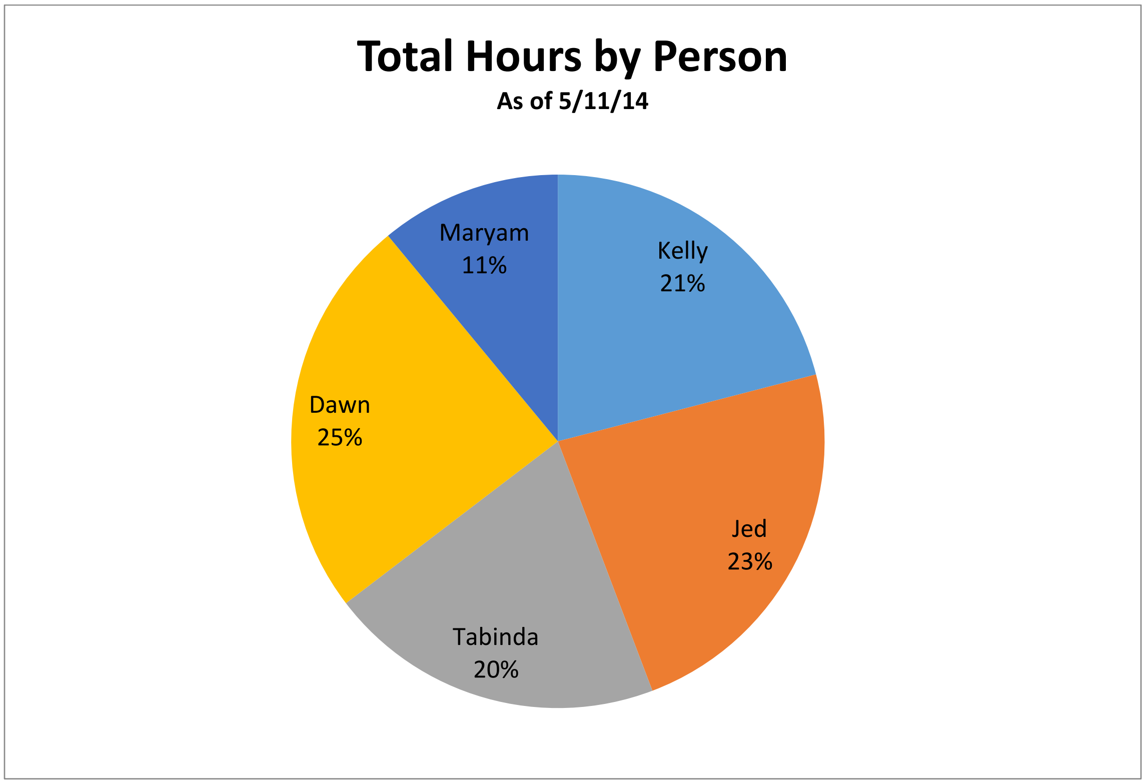


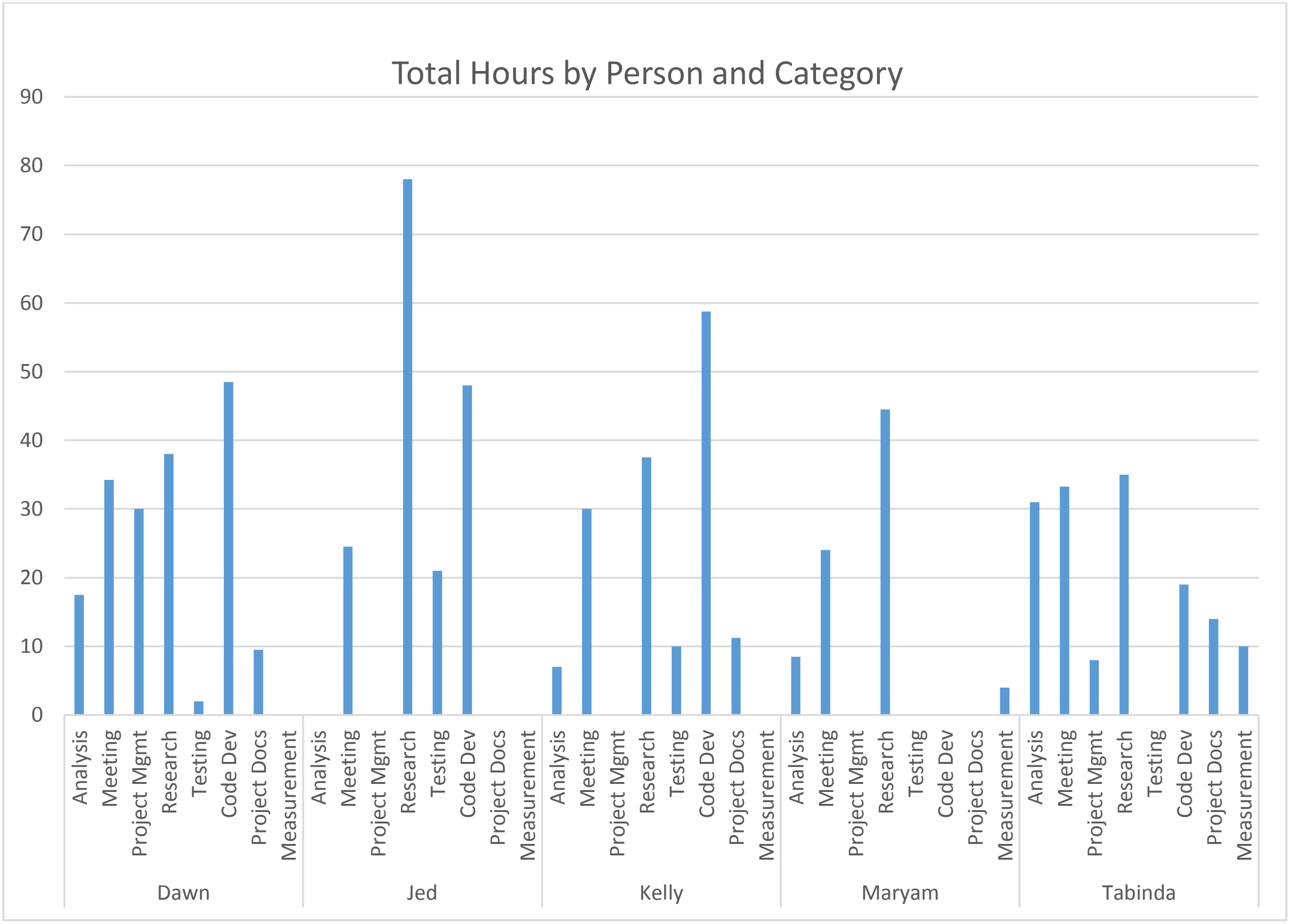



Total Hours by Category
As of 5/11/14
250
200
150
100
50
0
Analysis
Meeting
Project Mgmt
Research
Testing
Code Dev
Project Docs
Measurement


Total Hours by Category
As of 5/11/14
Testing 4%
Code Dev 24%
Project Docs 5%
Measurement 2%
Analysis 9%
Meeting 20%
Project Mgmt 5%
Research 31%


Total Hours by Person
As of 5/11/14
200
180
160
140
120
100
80
60
40
20
0
Kelly
Jed
Tabinda
Dawn
Maryam


Total Hours by Person
As of 5/11/14
Maryam 11%
Kelly 21%
Dawn 25%
Jed 23%
Tabinda 20%


Total Hours by Person and Category
90
80
70
60
50
40
30
20
10
0
Analysis
Meeting
Project Mgmt
Research
Testing
Code Dev
Project Docs
Measurement
Dawn
Jed
Kelly
Maryam
Tabinda

Final System Design

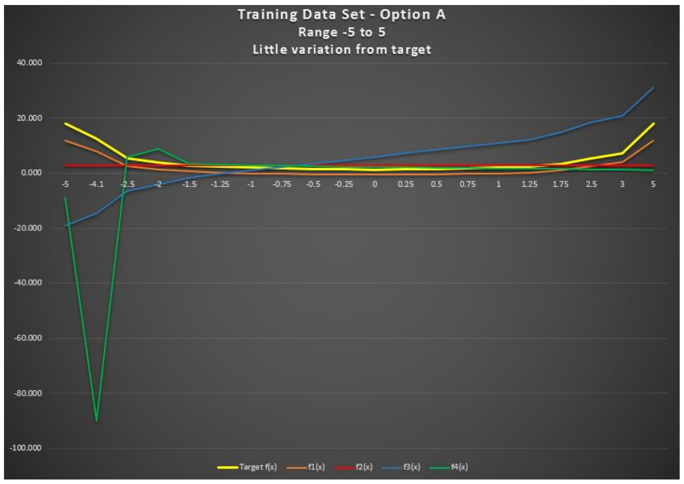
Selecting Training Data Set

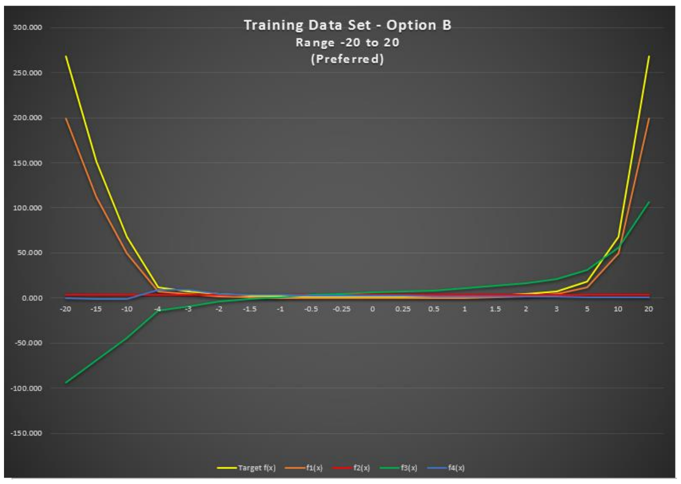
We decided our training data set would be comprised of 20 elements. When selecting the range of the "x" values, we considered two options:

Option A: $-5 \le x \le 5$
Option B: $-20 \le x \le 20$

We created a variety of test functions to measure fitness against our proposed data sets, e.g. a constant function, a quadratic function (with x^2 in the numerator), a function with x in the denominator. The resulting fitness values are shown below.

Because of the little variation we saw in our experimental fitness values with Option A, we chose to go with Option B, a range of x values between -20 and +20.



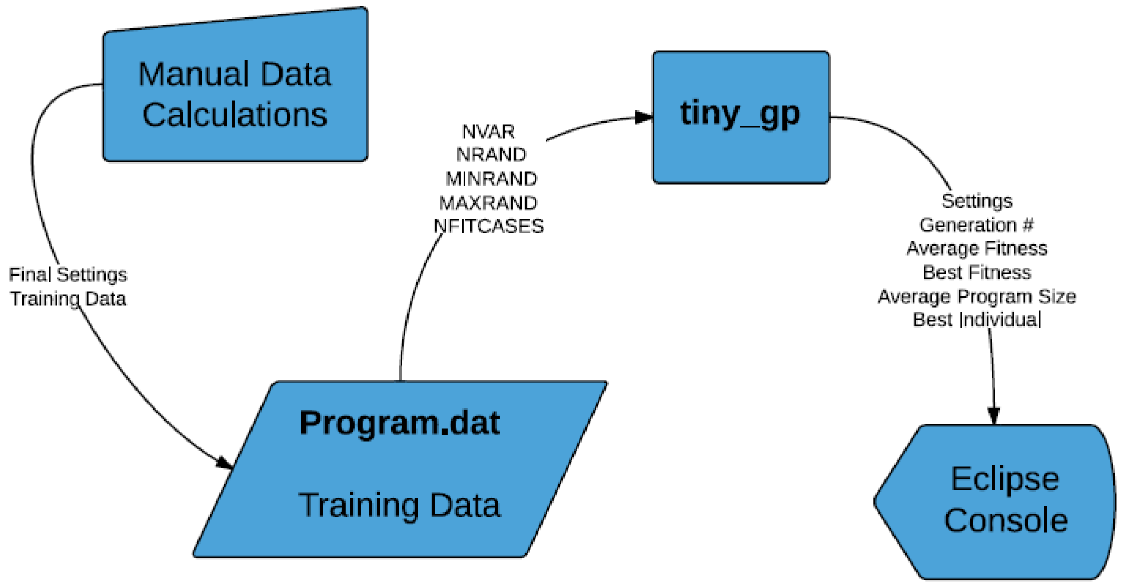


Settings

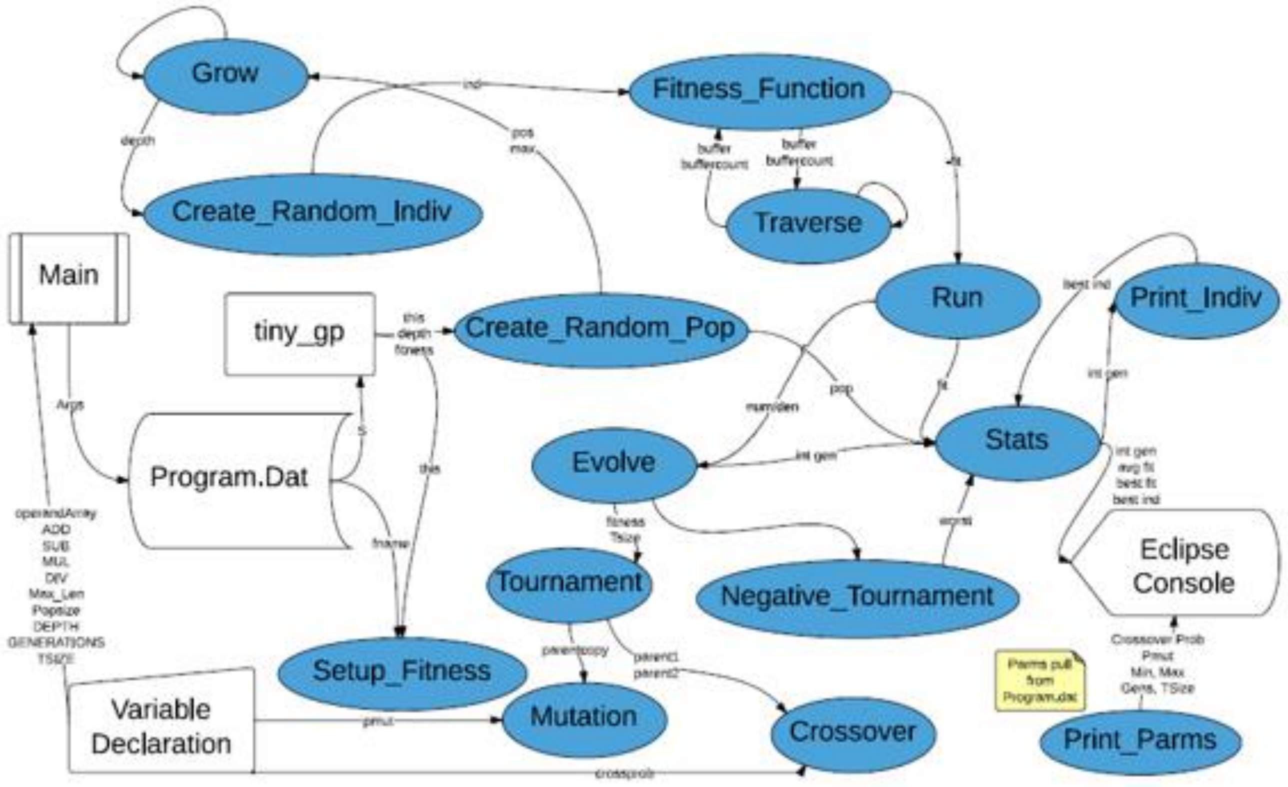
Initial Operands	0, 1, 2, 3, X (Used limited operands while developing initial code.)
Final Operands	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, X (Found it easy to expand set of operands in final code.)
Operators	+, -, *, /
Max Length	5
Tree Depth	3
Population Size	100,000
Tournament Size	4
Crossover Probability	0.90
Mutation Probability	0.05
Stop Program Criteria:	Fitness <= 0.001
	15 minutes
	100 generations

Data Flow Diagram

DFD Level 0

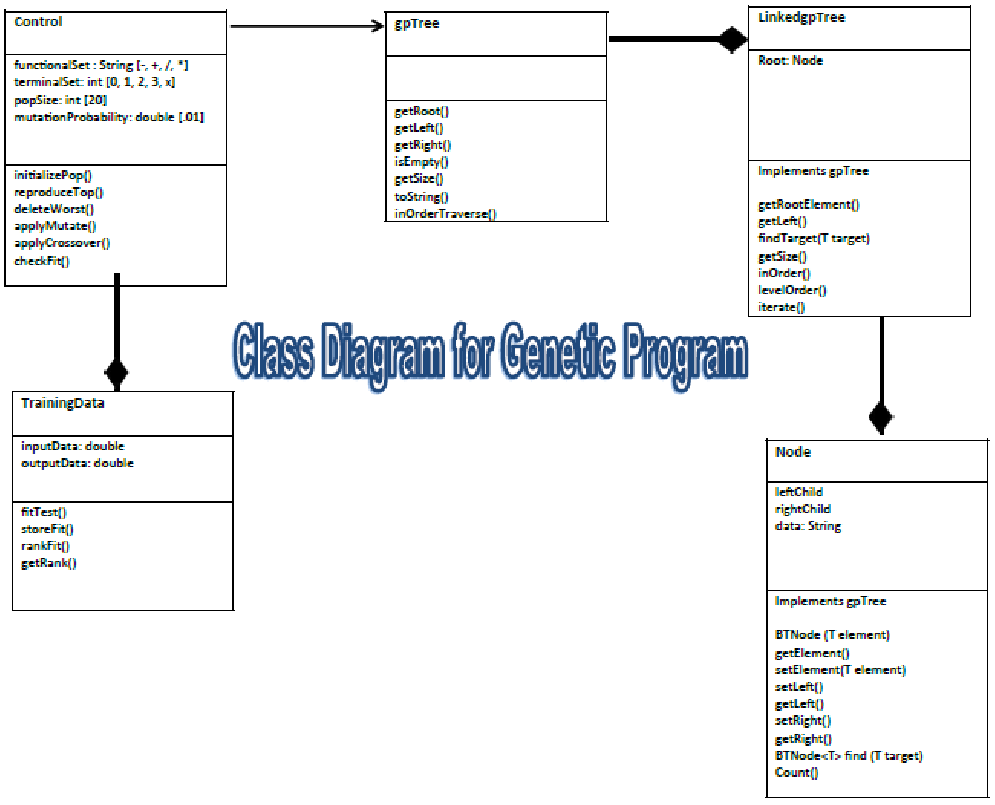


DFD Level 1

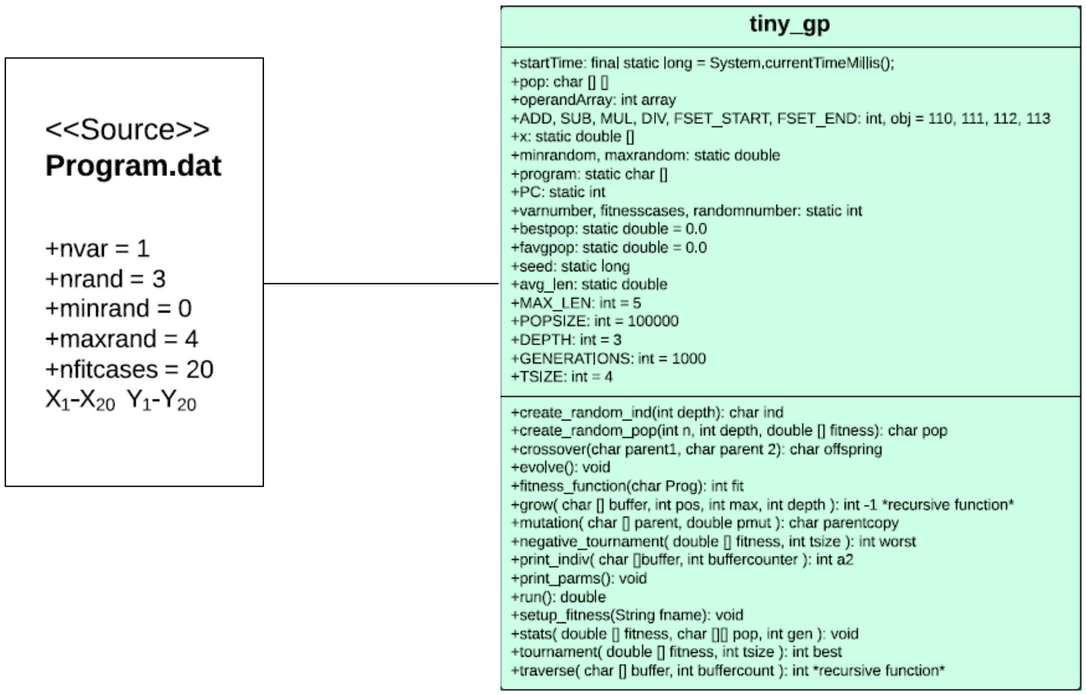


UML

Original UML



Final UML


<<Source>>
Program.dat
+nvar = 1
+nrand = 3
+minrand = 0
+maxrand = 4
+nfitcases = 20
X1-X20 Y1-Y20
tiny_gp
+startTime: final static long = System.currentTimeMillis();
+pop: char [] []
+operandArray: int array
+ADD, SUB, MUL, DIV, FSET_START, FSET_END: int, obj = 110, 111, 112, 113
+x: static double []
+minrandom, maxrandom: static double
+program: static char []
+PC: static int
+varnumber, fitnesscases, randomnumber: static int
+bestpop: static double = 0.0
+favgpop: static double = 0.0
+seed: static long
+avg_len: static double
+MAX_LEN: int = 5
+POPSIZE: int = 100000
+DEPTH: int = 3
+GENERATIONS: int = 1000
+TSIZE: int = 4
+create_random_ind(int depth): char ind
+create_random_pop(int n, int depth, double [] fitness): char pop
+crossover(char parent1, char parent 2): char offspring
+evolve(): void
+fitness_function(char Prog): int fit
+grow(char [] buffer, int pos, int max, int depth): int -1 *recursive function*
+mutation(char [] parent, double pmut): char parentcopy
+negative_tournament(double [] fitness, int tsize): int worst
+print_indiv(char []buffer, int buffercounter): int a2
+print_parms(): void
+run(): double
+setup_fitness(String fname): void
+stats(double [] fitness, char [][] pop, int gen): void
+tournament(double [] fitness, int tsize): int best
+traverse(char [] buffer, int buffercount): int *recursive function*

Division by Zero

Code line to handle a fraction with denominator close to 0:

```
        if ( Math.abs( den ) <= 0.001 ) return( num );
        else return( num / den );
```

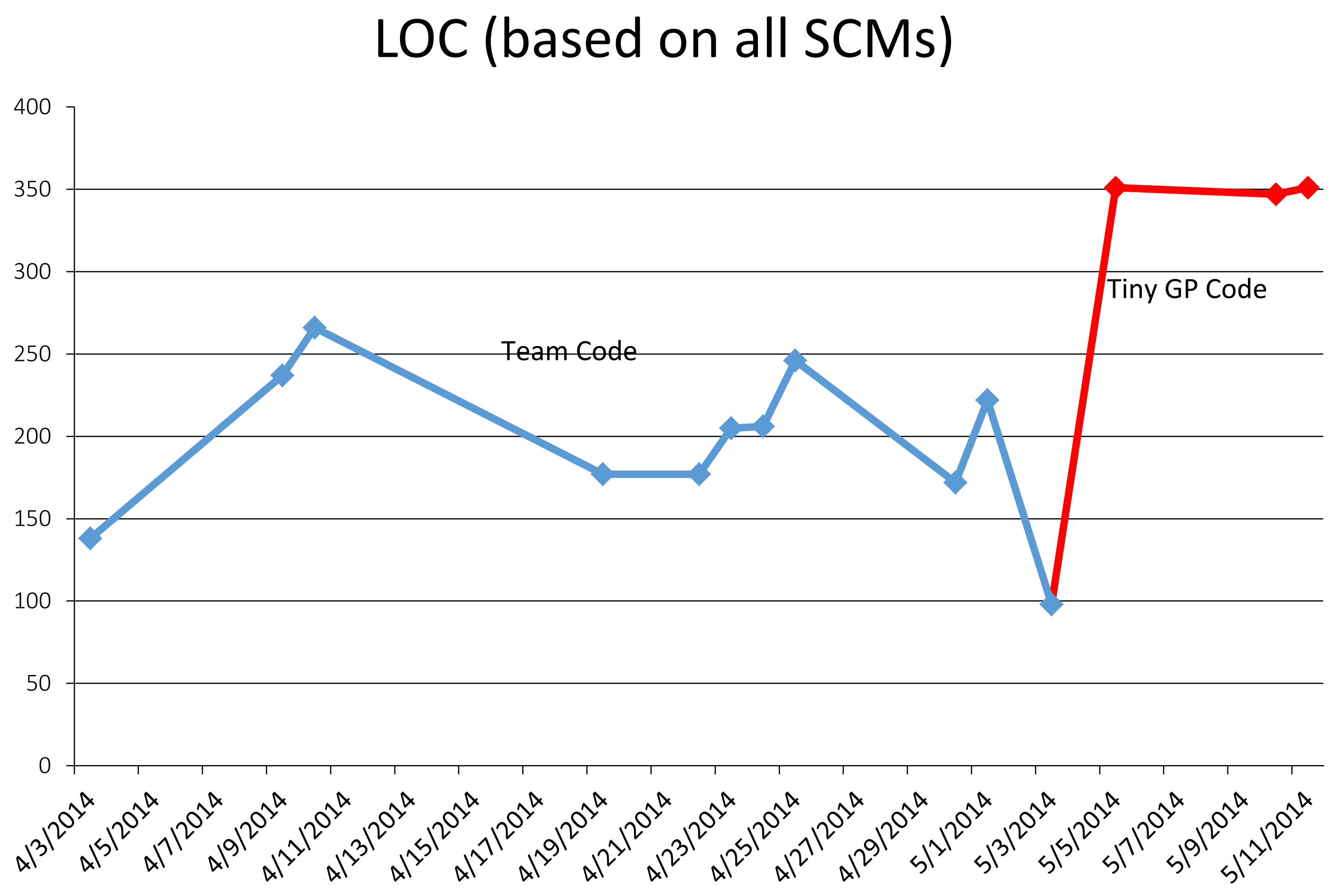
Translation:

If the denominator of a fraction is very small, (close to zero or equal to zero), return the numerator.

This is the equivalent of dividing the numerator by 1.

Measurements

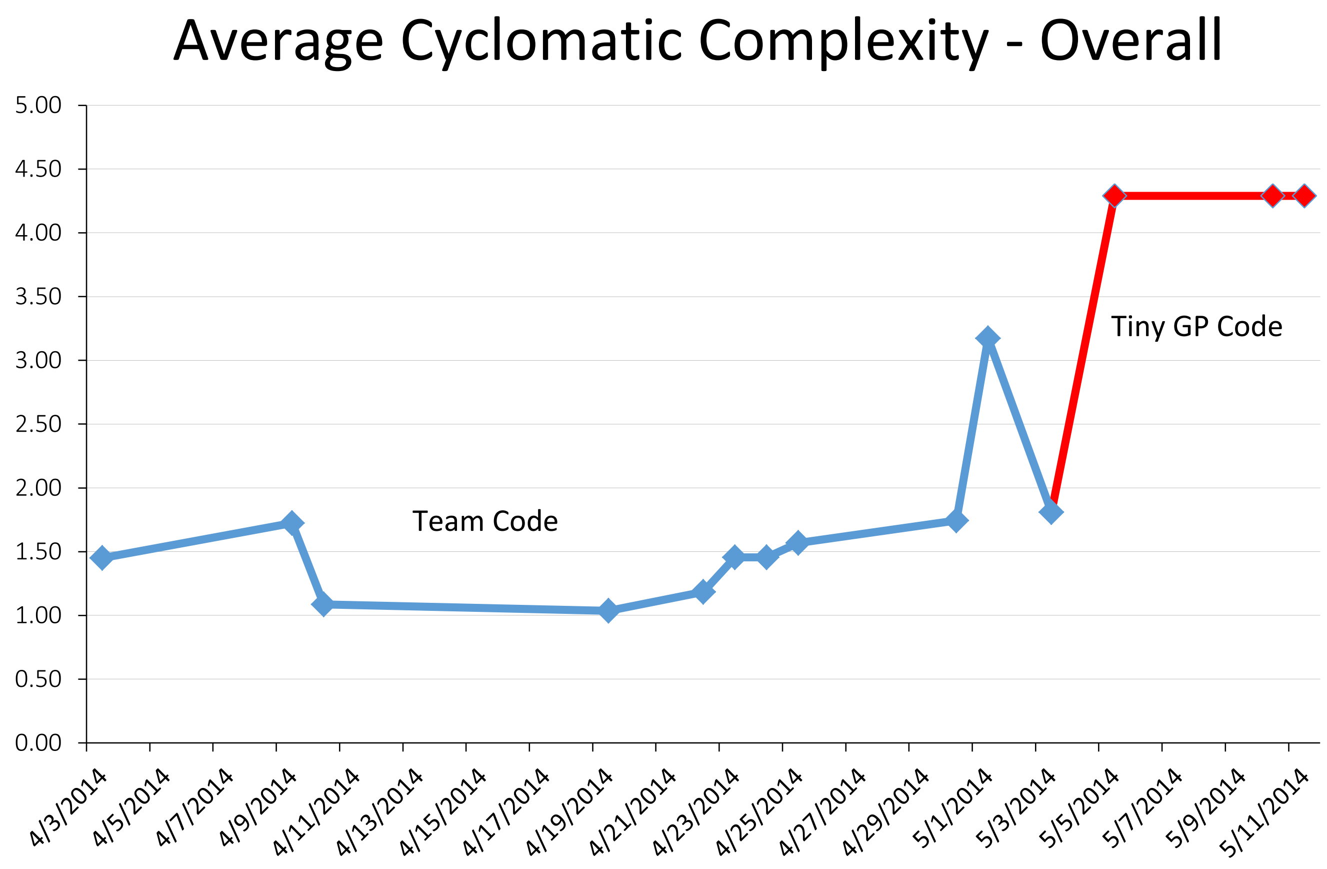
Lines of Code (LOC)



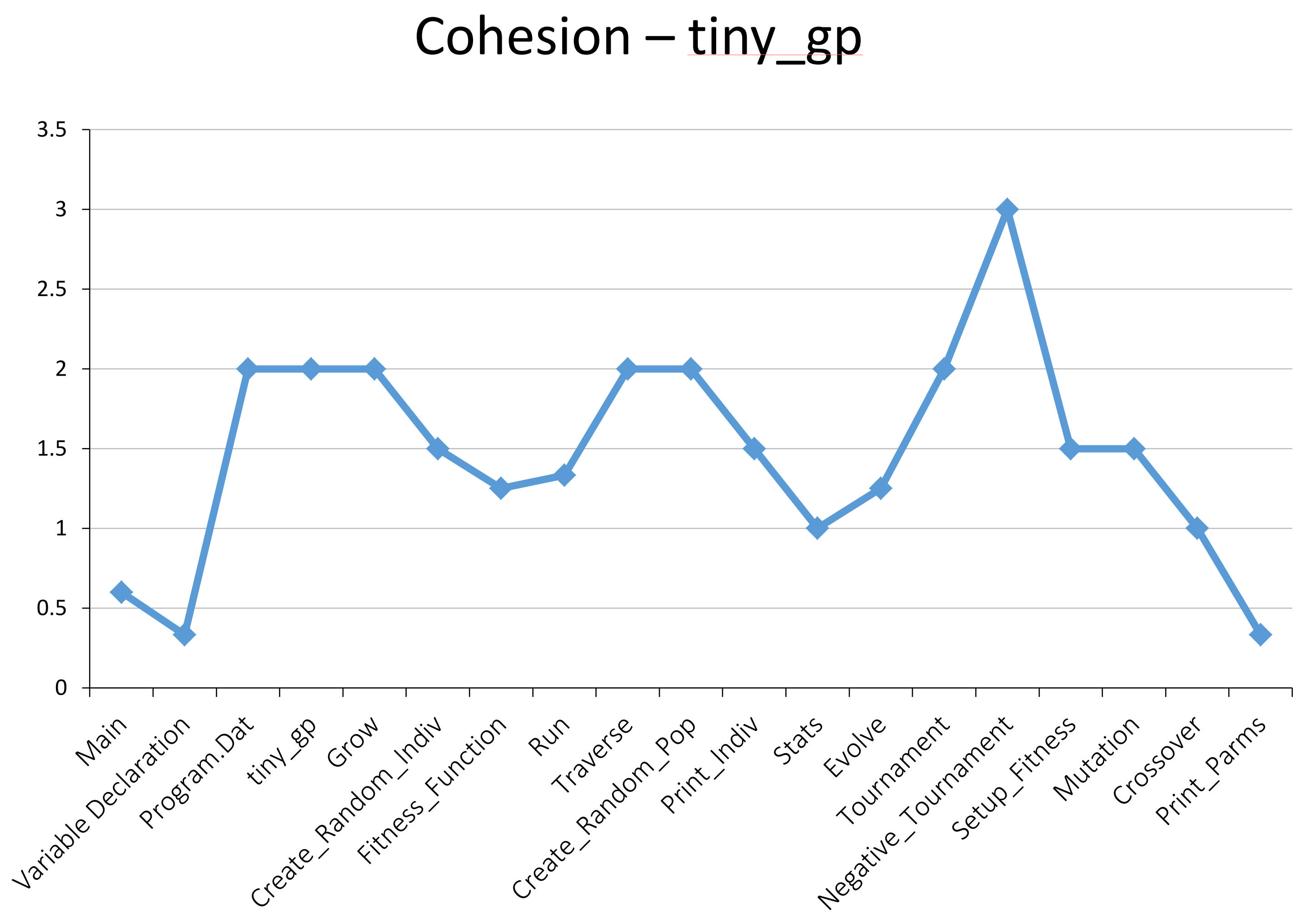
Cyclomatic Complexity

Average Cyclomatic Complexity – Overall

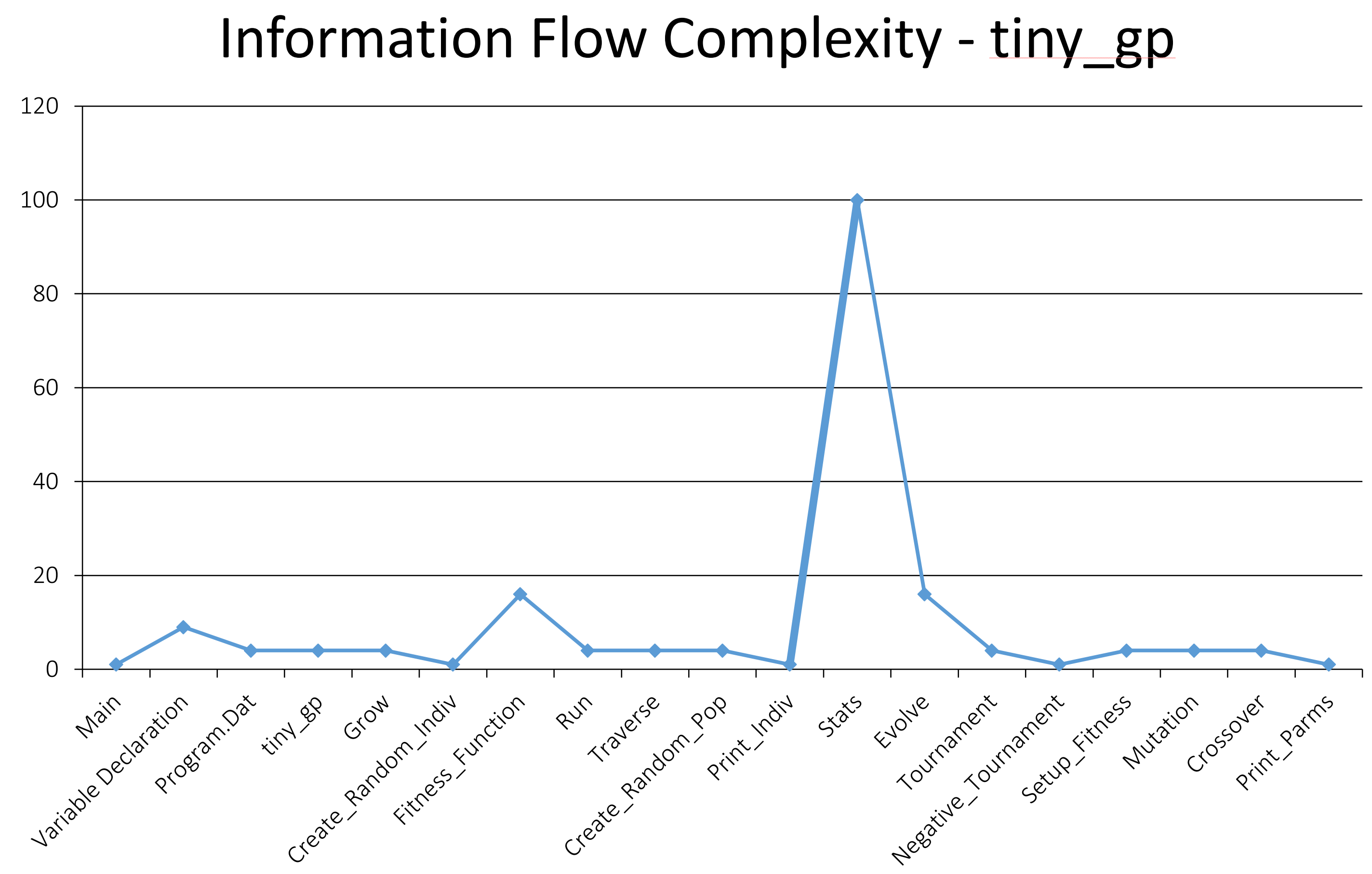
Date:	4/3	4/9	4/10	4/19	4/22	4/23	4/24	4/25	4/30	5/1	5/3	5/5	5/10	5/11
Avg CC:	1.45	1.73	1.09	1.04	1.19	1.46	1.46	1.57	1.74	3.17	1.81	4.29	4.29	4.29



Cohesion


Cohesion – tiny_gp
3.5
3
2.5
2
1.5
1
0.5
0
Main
Variable Declaration
Program.Dat
tiny_gp
Grow
Create_Random_Indiv
Fitness_Function
Run
Traverse
Create_Random_Pop
Print_Indiv
Stats
Evolve
Tournament
Negative_Tournament
Setup_Fitness
Mutation
Crossover
Print_Parms

Information Flow Complexity (IFC)



Relative System Complexity (RSC)

Methods	**Variables**	**FI [FanIn]**	**FO [FanOut]**	**Information Flow Complexity [IFC]**	**Functionality Complexity of each Method [H]**	**Cohesion [1/H]**
Main	5	1	1	1	1.67	0.6
Variable Declaration	12	-	3	9	3	0.33
Program.Dat	2	1	2	4	0.5	2
tiny_gp	2	1	2	4	0.5	2
Grow	2	2	1	4	0.5	2
Create_Random_Indiv	2	1	1	1	0.67	1.5
Fitness_Function	4	2	2	16	0.8	1.25
Run	3	1	2	4	0.75	1.33
Traverse	2	2	1	4	0.5	2
Create_Random_Pop	2	1	2	4	0.5	2
Print_Indiv	2	1	1	1	0.67	1.5
Stats	8	5	2	100	1	1
Evolve	4	2	2	16	0.8	1.25
Tournament	2	1	2	4	0.5	2
Negative_Tournament	1	1	1	1	0.33	3
Setup_Fitness	2	2	-	4	0.67	1.5
Mutation	2	2	-	4	0.67	1.5
Crossover	3	2	-	4	1	1
Print_Parms	6	-	1	1	3	0.33
				186	18.02	

Coupling, U (Sum of IFCs)	186.00
Total Functionality Complexity, H	18.02
Total System Complexity, C	204.02
Relative System Complexity, RSC	10.74

RSC <= 26 is good!

Additional Measurements

In addition to the charts and graphs provided above, there is much more measurement and analysis located in our Google docs **GP group project > Measurement > Measurements**. Tabinda performed measurement analytics on every class on each day that code was uploaded. Link:
https://drive.google.com/?usp=folder&authuser=0#folders/0BzpM7gg1JvssN0JudFFEc24xSUU

Testing

Summary

PART 1

The initial testing for our system revolved around 8 test cases written and executed by Jed. These test cases manipulated input data in the system to test how the program would react when different inputs were entered. Most of the test cases proved the flexibility of our program in that the program continued to function even with the changes. These test cases proved themselves valuable in allowing us to find which inputs caused the system to react in certain ways. One of the major areas of testing involved finding a function with a fitness value of 0 within our 15 minute time frame. Through our tests we believe we arrived at a synergistic balance which allows our system the perfect opportunity to find the best fitness value within the allotted time.

PART 2

The second aspect to our testing was the code coverage, also conducted by Jed. For this portion of the testing we used CodeCover, which is an open source tool that plugs into Eclipse and generates reports as you run your program. This tool was very easy to use. With a few simple steps Jed was able to get it running and generating reports for our system.

PART 3

The third aspect of our testing was Black box testing was used to confirm the program's ability to find the correct function (or an equivalent expression) in under 15 minutes. This testing, conducted by Kelly, was carried out by running the program with a random seed 30 times. Statistical data from each run was collected from the Eclipse console, and listed by category in an excel spreadsheet for further analysis. The results of this test are summarized by the following statistics:

- Correct function was found 30/30 times: a 100% success rate for this test sample.
- Minimum number of generations to solution: 2
- Maximum number of generations to solution: 14
- Average number of generations to solution: 5.1
- Fastest time to solution (seconds): 5.357
- Slowest time to solution: 23.275
- Average time to solution: 7.8726
- Average length of functions in final generation population: 13.336

Note: Test results were carried out on a Toshiba Satellite E45t-A4300 laptop

Part 1 – 8 Specific Tests

Test Case 1

Test Case ID: **TC_01_Correct Training Set_5-10-14**

Test Priority (Low/Medium/High): **Low**

Module Name:

Test Title: **Test Training data**

Description: **Test output with correct training data**

Test Designed by: **Jed Brundidge**

Test Designed date: **5-5-2014**

Test Executed by: **Jed Brundidge**

Test Execution date: **5-10-2014**

Pre-conditions: All inputs have been defined before system build

Dependencies: System data structure

Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Input training set	Example: 1,5, 0.9	Successful Run	Complete Fitness test	Pass	System acted as expected with correct inputs

Post-conditions:

The system has made a successful run. When the system is run successfully we will achieve 0 for our fitness.

Test Case 2

Test Case ID: TC_02_Double Test_5-10-14	**Test Designed by:** Jed Brundidge
Test Priority (Low/Medium/High): Low	**Test Designed date:** 5-5-2014
Module Name:	**Test Executed by:** Jed Brundidge
Test Title: Verify only type double is accepted for training input	**Test Execution date:** 5-10-2014

Description: Test that exception is thrown when no integer

Pre-conditions: Define training input.

Dependencies:

Step	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Input type Integer	Example: 1	System should throw exception	Exception	Pass	When integer is entered an error occurs
2	Input type Char	Example: 'A'	System should throw exception	Exception	Pass	When char is entered an error occurs
3	Input String	Example: “Hello”	System Should throw exception	Exception	Pass	When a string is entered an error occurs

Post-conditions:

None.

Test Case 3

Test Case ID: **TC_03_ChangeTreeDepth_5-10-14**

Test Designed by: **Jed Brundidge**

Test Priority (Low/Medium/High): Med

Test Designed date: **5-5-2014**

Module Name:

Test Executed by: **Jed Brundidge**

Test Title: **Change tree depth**

Test Execution date: **5-10-2014**

Description: **Test flexibility of tree depth. Change from 5**

Pre-conditions: Tree class built to specs

Dependencies:

Step	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Increase tree depth	Set depth to 15	Longer run time	Expected output received. System took longer to get the answer as expected	Pass	Larger depth means the trees are bigger.
2	Decrease tree depth	Set depth to 3	Shorter run time	Expected output received. System took shorter to get the answer as expected.	Pass	Shorter depth means tree is smaller.
3	Increase large	Set depth to 50	Extremely long run time	Did not finish. Run time took much longer than expected.	Pass	Very long to run with this depth
4	Increase small	Set depth to 2	Very short run time	Expected output received. System took a very short time to get answer as expected.	Pass	Very short.

Post-conditions:
Run time for each test went as expected. When the depth of the tree was decreased the program became less complex and took less time to find a solution. When the tree depth was increased the program became more complex making the run time longer.

Test Case 4

Test Case ID: **TC_4_Tree pop test_5-10-14**

Test Designed by: **Jed Brundidge**

Test Priority (Low/Medium/High): Med

Test Designed date: **5-5-2014**

Module Name:

Test Executed by: **Jed Brundidge**

Test Title: **Verify that trees have been created.**

Test Execution date: **5-10-2014**

Description: **Test population array.**

Pre-conditions: All size requirements must be in place.

Dependencies:

Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Create empty array list	Tree population	Non null array list	Populated array list	Pass	Trees are being populated correctly

Post-conditions: NA

Test Case 5

Test Case ID: **TC_5_Change Target Function_5-10-14**

Test Designed by: **Jed Brundidge**

Test Priority (Low/Medium/High): Med

Test Designed date: **5-5-2014**

Module Name:

Test Executed by: **Jed Brundidge**

Test Title: **Alter target function**

Test Execution date: **5-10-2014**

Description: **Determine program flexibility by altering the target function.**

Pre-conditions: **Program works successfully with initial target function.**

Dependencies:

Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Change + to -	Change operators in target function	Clean run	System found the closest match in under 15 minutes	Pass	System passed flexibility test

Post-conditions: NA

Test Case 6

Test Case ID: **TC_6_Population size_5-10-14**

Test Designed by: **Jed Brundidge**

Test Priority (Low/Medium/High): Med

Test Designed date: **5-5-2014**

Module Name:

Test Executed by: **Jed Brundidge**

Test Title: **Population test.**

Test Execution date: **5-10-2014**

Description: **Change population from 1000 to find best run time.**

Pre-conditions: This test is to find out how the system will react when the population size is either increased or decreased.

Dependencies: Population array.

Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Increase pop to 100000	Integer 10000	Increased run time	Problem solved quickly	Pass	Larger population is increased time
2	Decrease pop to 5000	Integer 500	Shorter run time	Problem solved slowly	Pass	

Post-conditions: NA

Test Case 7

Test Case ID: **TC_7_15 min run_5-10-14**

Test Priority (Low/Medium/High): Med

Module Name:

Test Title: **Length of run**

Description: **Change run parameters**

Test Designed by: **Jed Brundidge**

Test Designed date: **5-10-2014**

Test Executed by: **Jed Brundidge**

Test Execution date: **5-10-2014**

Pre-conditions: We need a fully functional program to test the time of each run when parameters are changed.

Dependencies: Max height of tree, Pop size

Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Increase max height	Increase max height of tree 15	Run will be longer than 15 min	Long run	Pass	
2	Decrease max height	Decrease max height to 3	Run will be very short	Short run	Pass	

Post-conditions: We can now see that the system is very flexible in allowing us to manipulate the max height of the trees to figure out how long our runs will take and allow us to find a sweet spot to find our target in less than 15 minutes.

Test Case 8

Test Case ID: TC_8_Mutation Test test_5-5-2014

Test Priority (Low/Medium/High): Med

Module Name:

Test Title: Test functionality of mutation

Description: Swap trees for mutation

Pre-conditions: Populate trees

Dependencies: Tree class and Population

Test Designed by: Jed Brundidge

Test Designed date: 5-10-2014

Test Executed by: Jed Brundidge

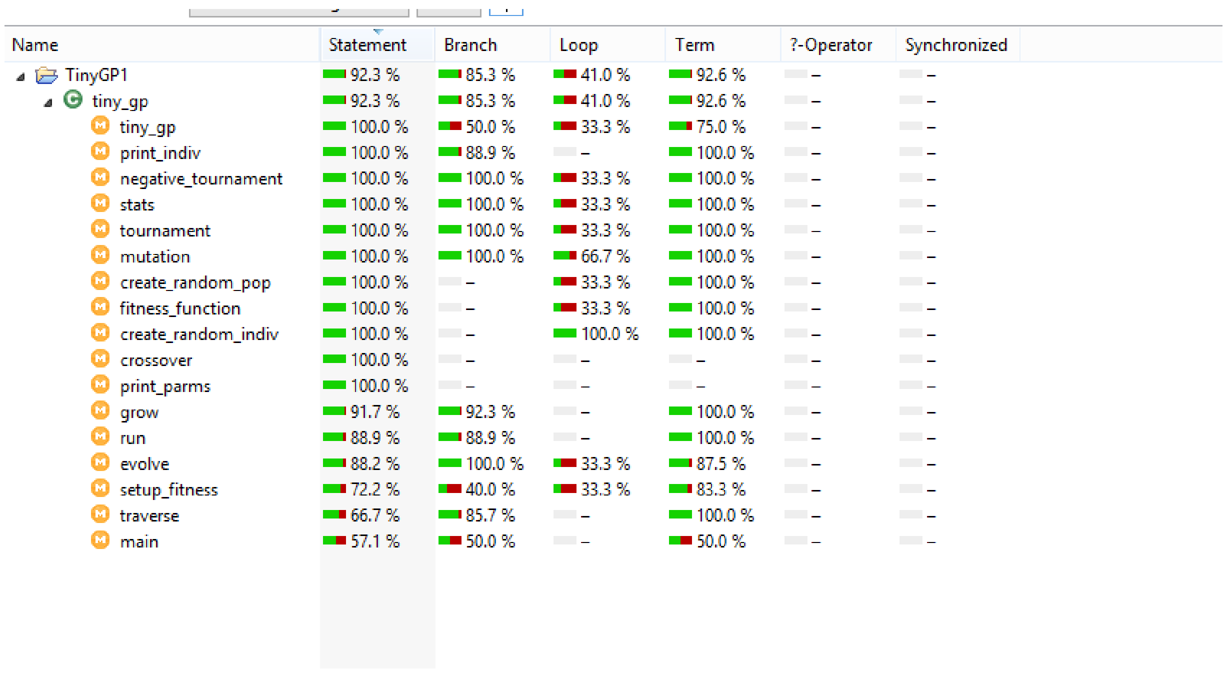
Test Execution date: 5-10-2014

Step	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status (Pass/Fail)**	**Notes**
1	Two trees with n nodes in each tree	First tree 3 + 2 / Second tree 1 + x +	Clean swap	Mutation was successful	Pass	

Post-conditions: NA

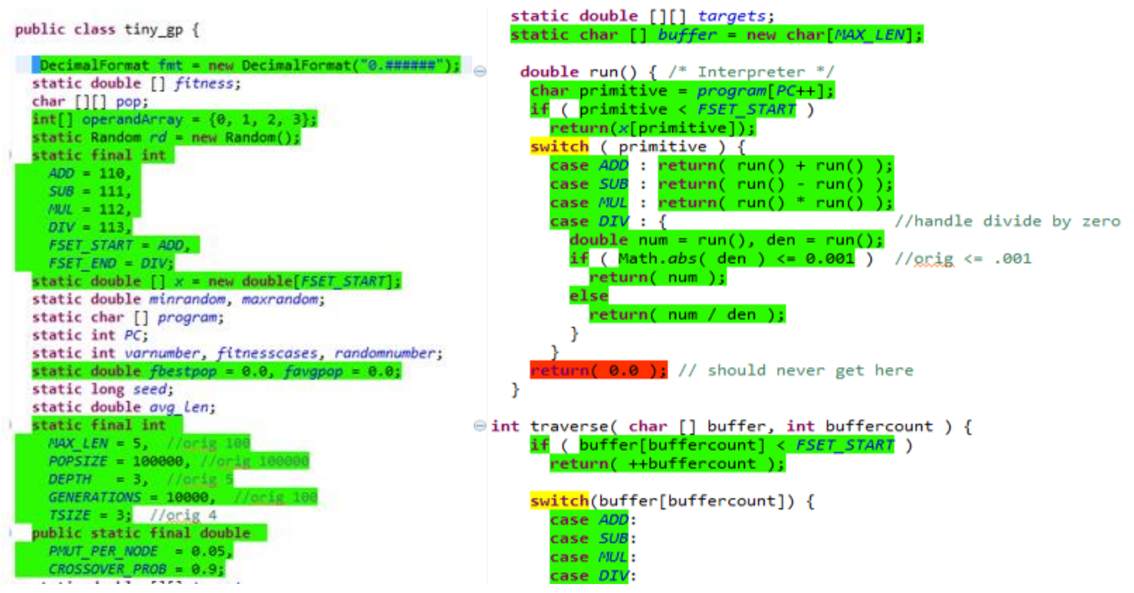
Part 2 – Code Coverage

The following display is from CodeCover application, used to test our tiny_GP program.



Name	Statement	Branch	Loop	Term	?-Operator	Synchronized
TinyGP1	92.3 %	85.3 %	41.0 %	92.6 %	–	–
tiny_gp	92.3 %	85.3 %	41.0 %	92.6 %	–	–
tiny_gp	100.0 %	50.0 %	33.3 %	75.0 %	–	–
print_indiv	100.0 %	88.9 %	–	100.0 %	–	–
negative_tournament	100.0 %	100.0 %	33.3 %	100.0 %	–	–
stats	100.0 %	100.0 %	33.3 %	100.0 %	–	–
tournament	100.0 %	100.0 %	33.3 %	100.0 %	–	–
mutation	100.0 %	100.0 %	66.7 %	100.0 %	–	–
create_random_pop	100.0 %	–	33.3 %	100.0 %	–	–
fitness_function	100.0 %	–	33.3 %	100.0 %	–	–
create_random_indiv	100.0 %	–	100.0 %	100.0 %	–	–
crossover	100.0 %	–	–	–	–	–
print_parms	100.0 %	–	–	–	–	–
grow	91.7 %	92.3 %	–	100.0 %	–	–
run	88.9 %	88.9 %	–	100.0 %	–	–
evolve	88.2 %	100.0 %	33.3 %	87.5 %	–	–
setup_fitness	72.2 %	40.0 %	33.3 %	83.3 %	–	–
traverse	66.7 %	85.7 %	–	100.0 %	–	–
main	57.1 %	50.0 %	–	50.0 %	–	–

This diagram demonstrates how CodeCover displayed the testing coverage of our code, line by line, in the Eclipse console. Green indicates full coverage and red indicates no coverage. The full picture is available on Google Docs: **GP group project > Testing > Testing700pm 5-18-14**. Link: https://drive.google.com/folderview?id=0B-sYVFgNOuZONkVLbzBuOXY0MGs&usp=sharing&tid=0BxP-i4iTYfT5LWtKcDFOU1VvbDQ



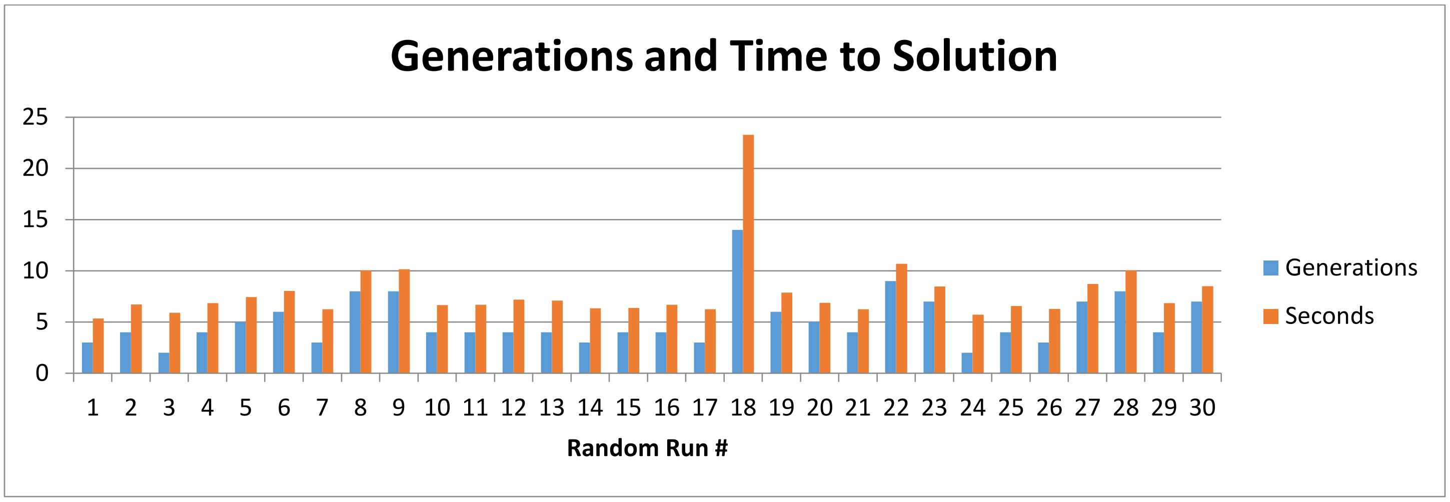
```java
public class tiny_gp {

    DecimalFormat fmt = new DecimalFormat("0.######");
    static double [] fitness;
    char [][] pop;
    int[] operandArray = {0, 1, 2, 3};
    static Random rd = new Random();
    static final int
      ADD = 110,
      SUB = 111,
      MUL = 112,
      DIV = 113,
      FSET_START = ADD,
      FSET_END = DIV;
    static double [] x = new double[FSET_START];
    static double minrandom, maxrandom;
    static char [] program;
    static int PC;
    static int varnumber, fitnesscases, randomnumber;
    static double fbestpop = 0.0, favgpop = 0.0;
    static long seed;
    static double avg_len;
    static final int
      MAX_LEN = 5,   //orig 100
      POPSIZE = 100000, //orig 100000
      DEPTH   = 3,   //orig 5
      GENERATIONS = 10000,  //orig 100
      TSIZE = 3;  //orig 4
    public static final double
      PMUT_PER_NODE  = 0.05,
      CROSSOVER_PROB = 0.9;
```

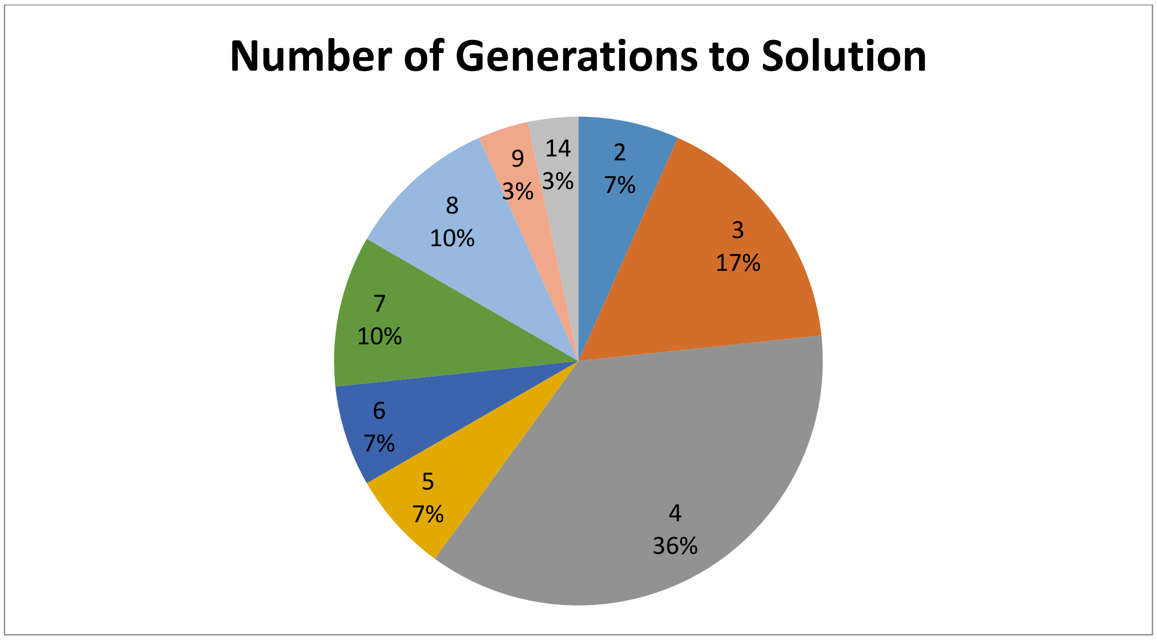
```java
  static double [][] targets;
  static char [] buffer = new char[MAX_LEN];

   double run() { /* Interpreter */
    char primitive = program[PC++];
    if ( primitive < FSET_START )
      return(x[primitive]);
    switch ( primitive ) {
      case ADD : return( run() + run() );
      case SUB : return( run() - run() );
      case MUL : return( run() * run() );
      case DIV : {                          //handle divide by zero
        double num = run(), den = run();
        if ( Math.abs( den ) <= 0.001 )  //orig <= .001
          return( num );
        else
          return( num / den );
        }
      }
    return( 0.0 ); // should never get here
  }

int traverse( char [] buffer, int buffercount ) {
    if ( buffer[buffercount] < FSET_START )
      return( ++buffercount );

    switch(buffer[buffercount]) {
      case ADD:
      case SUB:
      case MUL:
      case DIV:
```

Part 3 – Black Box Testing

Raw data for the following summary is located in Google docs: **GP group project > Testing > 30 Random Runs Output**. Link: https://drive.google.com/folderview?id=0BxP-i4iTYfT5UUppd19LZjlzcFU&usp=sharing&tid=0BxP-i4iTYfT5LWtKcDFOU1VvbDQ

Run #	# of Gens to Solution	Run time (seconds)	Final Gen Avg Size	Best Function
1	3	5.357	5.62092	((2.0 * 2.0) / (6.0 / (2.0 + (X1 * X1))))
2	4	6.719	5.48642	((X1 * X1) + (X1 + ((((4.0 / (X1 / X1)) - (X1 * X1)) / 3.0) - X1)))
3	2	5.912	4.33212	(2.0 * ((2.0 + (X1 * X1)) / 3.0))
4	4	6.848	5.47682	(((2.0 + (X1 * X1)) * 4.0) / 6.0)
5	5	7.452	8.77176	(((((X1 * X1) - 2.0) + 6.0) + (X1 * X1)) / 3.0)
6	6	8.036	12.02732	((((X1 * X1) - 4.0) / ((2.0 + 1.0) / 2.0)) + (4.0 / 1.0))
7	3	6.246	6.11244	((7.0 + (((X1 * X1) - 3.0) + (X1 * X1))) / 3.0)
8	8	10.039	21.53586	((2.0 + (X1 * X1)) - (((X1 * X1) + 2.0) / (8.0 - 5.0)))
9	8	10.155	25.54626	(4.0 / ((2.0 + 4.0) / ((X1 * X1) + 2.0)))
10	4	6.668	5.45922	((X1 * X1) - (((((X1 * X1) - (0.0 + ((1.0 * 0.0) - X1))) - X1) - 4.0) / (3.0 / 0.0)))
11	4	6.683	5.82284	((X1 * X1) - (((((X1 + (X1 * X1)) - X1) + (8.0 - 4.0)) - 8.0) / 3.0))
12	4	7.187	5.28024	((X1 * X1) - (((X1 * X1) - 4.0) / 3.0))
13	4	7.107	7.13632	((X1 * X1) - (((X1 * X1) / 3.0) - (8.0 / 6.0)))
14	3	6.33	4.87936	((X1 * X1) + ((4.0 - (1.0 * (X1 * X1))) / 3.0))
15	4	6.387	5.37576	((((8.0 - (((X1 * X1) - X1) - 8.0)) - (((X1 + (X1 * X1)) + 4.0) + 4.0)) / 6.0) + (X1 * X1))
16	4	6.697	5.9177	((X1 + ((2.0 + ((X1 + (X1 * X1)) - X1)) / (3.0 / 2.0))) - X1)
17	3	6.261	4.5918	((X1 * X1) - (((X1 * X1) - 4.0) / 3.0))
18	14	23.275	111.22718	(((0.0 - (X1 - X1)) / (X1 + (((X1 * 4.0) - ((((((7.0 / 5.0) + (((((X1 * 4.0) - X1) - X1) * X1) + 4.0)) / 4.0) / 8.0) - X1) - X1)) * (0.0 + (X1 * X1))))) + (((X1 * X1) + ((X1 * X1) + 4.0)) / (7.0 - 4.0)))
19	6	7.87	10.06368	((X1 * X1) + (X1 - ((((X1 * X1) + ((((X1 * X1) / 2.0) - ((4.0 - ((0.0 * X1) / (X1 + 4.0))) - X1)) - (((X1 * X1) / 2.0) - X1))) + X1) / 3.0)))
20	5	6.888	9.24008	((3.0 + (X1 * X1)) - (((((X1 + 3.0) * X1) - 1.0) / 3.0) + (2.0 - X1)))
21	4	6.247	5.4528	(((0.0 + ((X1 * X1) + 4.0)) + ((X1 * X1) + 0.0)) / 3.0)
22	9	10.687	29.1852	((2.0 + (X1 * X1)) / ((5.0 - 2.0) / 2.0))
23	7	8.457	14.72522	(((X1 * X1) + (8.0 / 4.0)) * (4.0 / 6.0))
24	2	5.724	4.07976	(2.0 * (((X1 * X1) + 2.0) / 3.0))
25	4	6.577	5.29842	((X1 * X1) - ((4.0 - (X1 * X1)) / ((X1 - 3.0) - X1)))
26	3	6.274	5.5149	(((4.0 - (X1 * X1)) / 3.0) + (X1 * X1))
27	7	8.73	22.27664	(((4.0 + (X1 * (X1 + X1))) / (8.0 - 5.0)) + 0.0)
28	8	10.026	22.58422	((X1 * X1) - (((X1 * X1) - ((4.0 + 4.0) - (X1 * X1))) / 6.0))
29	4	6.844	6.83886	(((7.0 - 3.0) + ((X1 * X1) + (X1 * X1))) / 3.0)
30	7	8.495	14.21442	((((X1 * X1) + 4.0) + (X1 * X1)) / (7.0 - 4.0))
AVG	5.1	7.8726	13.336	


Generations and Time to Solution
25
20
15
10
5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Random Run #
Generations
Seconds


Number of Generations to Solution
2 7%
3 17%
4 36%
5 7%
6 7%
7 10%
8 10%
9 3%
14 3%

Post Project Analysis

Summary

This genetic programming project was an excellent learning experience for all of us. By the end of the semester we had the tools needed to measure and test the project. However, throughout the semester we lacked the tools necessary to code this project ourselves.

We had three people coding different aspects of the program: Kelly working to generate random binary trees, Jed working to perform crossover and mutation to evolve the population of trees, and Dawn working training data, fitness testing, and ranking. We held onto our optimism until May 1st, when two weeks before the project was due, we didn't have any of the three sections of code working perfectly, and still faced the task of connecting the pieces.

As we shifted gears to find a package, we were rushed. Had we looked for a package sooner, we could have applied measurement software applications against various packages to make an objective decision, and we would have had more time to work with packages that we quickly eliminated because we couldn't modify or enhance to run for our application.

With our final product, a modification of a program called tiny_gp, we had it running for a few days, but could not get a successful outcome. Finally, we had a breakthrough on May 10th, two days before our presentation. We launched into testing and measurement as quickly as we could. The team pulled together, and we felt our final product and presentation were of good quality.

In the week that followed the presentation, we continued to work with tiny_gp, conducting 30 random, timed-tests and expanding the operand set. This additional information is reflected in the Testing section of this, our final report.

Lessons Learned

Time Measurement

It would have been beneficial to:
(a) Define our time categories more precisely from the beginning
(b) Report hours weekly rather than after long spans of time, and
(c) Allocate meeting hours toward specific time categories rather than toward a general "meeting" category.

Each one of our meetings revolved around a specific task such as analysis, coding, or document preparation as a group, as is evident in our meeting minutes. However, we could have given more time to research and analysis of the overall project "together as a team with everyone participating actively in discussions. This could have been achieved by attaching responsibilities to individuals with set timeline expectations earlier on so each person knew what was expected of them and when... rather than do what you can. Our measurement of time-spent is not as accurate as it should be due to weakness in these areas.

Testing

We should have shifted our pre-conceived ideas about testing only the end product (pass or fail), to methodically testing our work in progress so that we could optimize our final product result. For example, we did not record test results as we modified our package code. We changed settings like population size, tree depth, etc. to achieve a better end result, but had no pattern or record of best combination of settings. It would have been beneficial to use a methodical plan *to* record those in-process tests.

Package Selection

For package selection we should have (a) started the search sooner, (b) used software measurement applications to compare the package candidates to one another, (c) established weighted criteria for what we would want in a package, and (d) graded each package in each weighted category so that we could assign a numerical/objective value to each package. Subjectivity would still have been evident in the steps of weighting of the criteria and grading, but the other steps in the process would lend some objectivity to the selection process.

Sometimes when you get stuck in code development, it's a sign you need to scrap what you are working on and start over (as painful as that is). Our developer, Kelly Heitz, spent a significant portion of her coding hours attempting to get a recursive insert method working on our binary trees in level order. The original code contains multiple attempts, all commented out with a description of what each attempt did instead. It was only when she "gave up" and began working on the Tiny GP program that Heitz felt she finally understood how nodes could be added to the tree in the fashion desired.

Measurement

We learned how to use tools to do measurement, which greatly improved our team's efficiency toward the end of the project. In order to implement accurate measurements of our code, we searched for and implemented CodeAnalytix as a shared tool. This plug-in allowed us to apply lessons learned in class regarding code cohesion and complexity.

Task Prioritization

Since we became stuck trying to write our own code, our team learned how to prioritize as a group. We worked collaboratively to ensure that tasks of the highest priority were accomplished first, and then worked as possible on additional items that we thought would be beneficial (rather than necessary) within our final program and report.

Software Configuration Management (SCM)

The historical record and evolution of all of the code we developed and our final modified package code for tiny_gp are located as .txt files on Google Docs: **GP group project > SCM code files**. Link:

https://drive.google.com/folderview?id=0BzpM7gg1JvssZkdNd3hpX2hPOTQ&usp=sharing&tid=0BxP-i4iTYfT5LWtKcDFOU1VvbDQ

User Manual

Tiny_GP Program User Manual

Pre-Installation:

There are two ways to run our Genetic Program:

- **Integrated Development Environment**
  *Eclipse was used for the development of the final product and is the recommended IDE:

  Download Eclipse from: https://www.eclipse.org/downloads/

- **Command Prompt**
  * Note: Both run methods require installation of the Java Development Kit as a prerequisite.

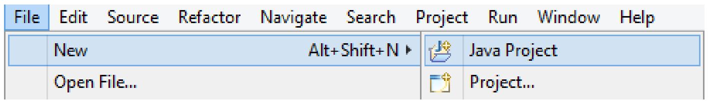
  Download JDK from: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Installation:

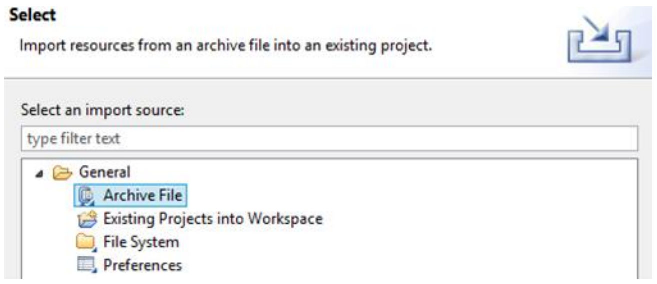
1. Download the tiny_gp folder. Link to this folder:

If using Eclipse as your IDE:

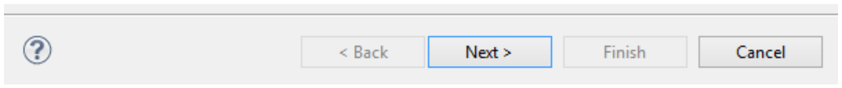
1. Click the **File** tab, select **New**, and create a **Java Project** in Eclipse.



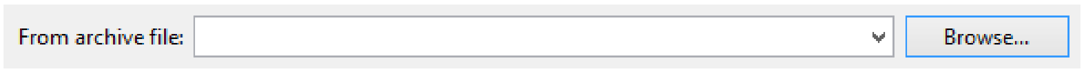
2. Right click on your project. A drop down menu will appear.
3. Select **Import...** from the list of available options.
4. In the pop-up import window which appears, select the following:
   a. Expand the **General** file
   b. Select **Archive File**


Select
Import resources from an archive file into an existing project.
Select an import source:
type filter text
General
Archive File
Existing Projects into Workspace
File System
Preferences

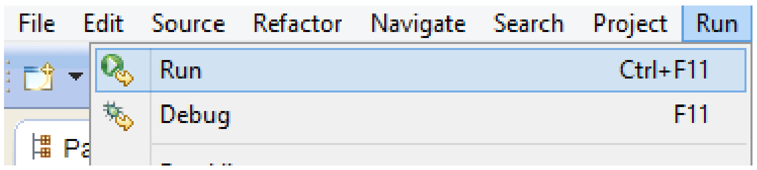
5. Click **Next** at the bottom of the import pop-up screen.



6. Select the **Browse** button to open a file explorer.



7. Select the tiny_gp folder from the download location.
8. Run the project by clicking the Run tab, and selecting Run (the first option in the drop down menu).



If using the Command Line:

PC

1. Open the Command Prompt window by clicking the **Start** button.
2. Click **All Programs.**
3. Select **Accessories**.
4. Select **Command Prompt**.

Mac

1. Open **Finder**. (Finder is available in the Dock.)
2. Select **Applications**, then chose **Utilities**.
3. Double click on **Terminal**.

Run the Program

1. Type the directory location where you saved the tiny_gp download.
2. Append “dir” to the directory name to show the contents to ensure tiny_gp.java is included as a file.
3. Enter “javac tiny_gp.java” to compile the code.
4. Enter “java tiny_gp” to run the program.