



EVOLUTIONARY COMPUTING SYSTEM

A software project inspired by Darwin's theory of evolution

July 2016

Authored and Developed by:

Maverick Edberg
Anders Skaar
Muntaser Khan
Brett Branan
Aghogho Ometan

SEIS 610: Software Engineering
Graduate Programs in Software Engineering
University of St. Thomas

Table of Contents

Abstract.....	3
Introduction	4
Evolutionary Computing	4
Project Description.....	4
Requirements Analysis.....	5
Technical Requirements.....	5
Configuration Requirements.....	6
Performance Requirements.....	6
User Experience Requirements.....	7
System Design	8
Program Location & Language Selection	8
Software Architecture Approaches.....	8
Expression Trees (Binary).....	8
Mutation	9
Crossover	9
Evolution	9
Deep Cloning.....	9
Interface vs. Abstract Class	10
Factory Pattern	10
Comparator vs. Comparable	10
Enums.....	10
Custom Exception Classes.....	11
Final Classes	11
Configuration	11
Population Size.....	11
Expression Tree Height	11
Target Fitness Value.....	11
Max Run Time	11
System Testing	12
Whitebox Testing	12
Unit Testing:.....	12
Code Coverage:	13

Blackbox Testing:	14
Data Input / Output Test.....	14
Data Structure Test	14
Metrics	14
Complexity Metrics	15
Cyclomatic Complexity.....	15
Weighted Method Complexity.....	16
Dependency Metrics	17
Cyclic Dependencies.....	17
Dependencies.....	17
Dependents.....	18
Lines of Code Metrics.....	19
Lines of Code.....	19
Cohesion Metrics	20
Lack of Cohesion	20
Project Results	21
Bibliography	22
Appendices:.....	23
Appendix A: Software User Manual.....	24
Appendix B: Diagrams & Visuals	26
Dataflow Diagram:	26
Finite State Machine:	27
Object Oriented Model:	28
Appendix C: Post-Project Analysis	29
Lessons Learned:	29
Future Work	29
Group Collaboration Tools (Pros/Cons, Decisions)	29
Group Member Assignments and Time Allocation	32
Appendix D: Weekly SCM Files / Meeting Notes	33

Abstract

As a whole our project was successful. Our evolutionary computing system was capable of finding an exact solution to the provided function (Fitness value of 0). Our code was structured in such a way that we could easily accommodate other functions (in case the requirements change one week before final delivery), and do so without concern for inducing bugs, changing performance, or losing any other functionality. Our group succeeded in meeting its own individual goals on all levels, including cross platform compatibility, and a GUI allowing user to adjust input parameters and other settings. Our project functional accuracy, reliability, maintainability, were verified quantitatively by measuring and recording all key software measures and metrics, and also performed a full suite of software tests ensuring sufficient test methods, quantity, and quality for maximum coverage.

Introduction

Evolutionary Computing

Evolutionary computation is a particular subfield of artificial intelligence and software engineering which use techniques that mimic Darwin's principles evolution. In order to solve a problem, a certain set of potential solutions are tested in an iterative/cyclical process. Using the results (fitness values) from every test iteration as a measure of success/accuracy, the system is able adjusts certain variable or "evolves" until it has found the ideal (or more optimal) solution.

Project Description

The evolutionary computing system as defined in the SEIS 610 project deliverable was fairly straightforward – The objective of the project was to develop an Evolutionary Computing software program which is capable of generating a solution which is equivalent to a simple 2-variable function. The software would need to generate a solution by using a set of training data (x and y inputs) and calculating (output) fitness values for each iteration until it finds the lowest possible - or ideally - the exact solution with a fitness value of 0.

Requirements Analysis

Technical Requirements

An initial set of technical requirements has been given to the project teams through class content and lectures. Using these requirements our team built out an evolutionary computing system prototype. This prototype found solutions for given training sets, but was inconsistent in both success and performance. Through trial, error, lecture learnings, and an incremental process, our team determined the remaining necessary technical requirements for our evolutionary computing system:

Requirement Type	Requirement
Technical (Code)	Must be able to evaluate the fitness of a given solution
Technical (Code)	Integer Operands (leafs) must be able to mutate into a random integer
Technical (Code)	Variable Operands (leafs) do not need to be mutated
Technical (Code)	Operators (branches) must be able to mutate into a random operator
Technical (Code)	Must be able to crossover two individuals
Technical (Code)	Must be able to randomly mutate individuals
Technical (Code)	Mutations should not cross types (i.e. an operand to an operator)
Technical (Code)	Must be able to determine the two most fit individuals for crossovers
Technical (Code)	Top two fitness individuals are brought into the next generation as-is
Technical (Code)	Top two fitness individuals are crossed over and both children are brought into the next generation
Technical (Code)	All individuals that are not the top two or the cross-overs from the previous generation are mutated
Technical (Code)	After one thousand generations of failed solutions the entire population is reset
Technical (Code)	When crossing over two individuals we cannot exceed the max tree height
Technical (Code)	Min tree height is 3
Technical (Code)	Operands must include {0-9} and a single variable x
Technical (Code)	Operators must include addition, subtraction, multiplication, and division
Technical (Code)	Evolutionary process must stop once a solution with a fitness of target fitness has been found or the time limit has been reached

Configuration Requirements

Initially the only configuration requirement given to the project teams was the requirement that the evolutionary computing system must take a training set as an input to solve. As we developed technical requirements, however, our team identified several other configuration requirements that help to stabilize and improve the performance of our evolutionary computing system:

Requirement Type	Requirement
Configuration	Must be able to take fitness set as input
Configuration	Fitness key value pairs are delimited by new lines, while key and value are delimited by a semicolon (e.g.: 12;456)
Configuration	Must detect and error on malformed fitness set input
Configuration	Must detect and error on empty fitness set input
Configuration	Must be able to take population size as input
Configuration	Population size input has a min of 7 and max of 20
Configuration	Must be able to take max tree height as input
Configuration	Max tree height input has a min of 3 and a max of 5
Configuration	Must be able to take target fitness value as input
Configuration	Target fitness value input has a min of 0 and a max of 100
Configuration	Must be able to take maximum run time as input in minutes
Configuration	Maximum run time input has a min of 1 minute and a max of 15 minutes

Performance Requirements

Project teams were given one performance requirement at the beginning of the project. Throughout the project our team did not identify any new performance requirements worth noting. As stated - This evolutionary computing program must be able to generate target function $(2x^2-2)/3$ in 15 minutes. The fitness value is provided by the user. The target function listed did change in a last minute requirement adjustment, however our technical requirement that states training sets must be read from a file made adapting to this change very easy. Our training set file was simply changed and we were done. Nonetheless, our one performance requirement is:

Requirement Type	Requirement
Performance	Must complete in 15 minutes (1200s) or less

User Experience Requirements

Our user experience requirements were to simply make the software as user-friendly as possible. To simplify use, we determined that we would use a GUI to run the software rather than a CMD line interface. The GUI requirements which were established are as follows:

Requirement Type	Requirement
User Experience	Must have a start button to begin the evolutionary process
User Experience	Must have a cancel button to stop the evolutionary process
User Experience	Must show how many generations have been evaluated (not counting reset population generations)
User Experience	Must show how long the evolutionary process took
User Experience	Once the solution runs out of time the main menu should be displayed
User Experience	Should show a graph of the fitness of the best solutions found over time after successful completion (not out of time or cancelled)
User Experience	Inputs must be appropriate for the given data type, i.e. textboxes, combo boxes, etc.
User Experience	Dialogs must be modal when necessary to direct user attention (i.e. to the output graph or to the cancel button etc.)
User Experience	Dialogs must be used only when providing output or additional input to prevent unnecessary clicks or redirection

System Design

Program Location & Language Selection

Our team's EC System Application resides locally on all team members' laptops. It is saved on Gitlab as a form of source control. All developers can create separate feature branches and work concurrently.

Our team built our EC System application from scratch which doesn't use any third party applications or libraries for any of its core evolutionary features. During the technology selection process we had to choose between a handful of solutions that our team members identified: A custom Java solution, JGAP, JGP, ECJ, Jenetics, Watchmaker, and Jenes. All of these technologies had their own pros and cons, however the following table helped us determine that a custom Java solution met our requirements the best:

	JGAP	JGP	ECJ	Jenetics	Watchmaker	Jenes
Programing Language	Java	Java	Java	Java	Java	Java
Interface Type	Command	No	Command	Command	No	Command
Tree	No	Yes	Yes	Yes	Yes	Yes
Output	Command	No	Files	GUI	GUI	Files
Code Access	Yes	Limited	Yes	Yes	GUI	Yes

Our Development end date was July 16,2016. Testing took place at the same time as development. We worked on small portions of our deliverable and tested it in a sudo-agile way. As a whole, our development followed an incremental approach with small pieces of functionality being added over time after development of the core functionality. This was a great approach since we were able to build a working prototype far before having a complete set of requirements.

Software Architecture Approaches

During the development process we came across several development approaches that fit our requirements and our desired solution. We outlined some of these and documented why they were preferred over any possible potential alternatives:

Expression Trees (Binary)

Our team determined very early on that binary trees were the correct data structure to use in this application. In fact, a term called "Expression Tree" already exists since this is an already popular use case of binary trees. The reasons why binary trees are the preferred approach are:

- Ordered structure
- Simple recursive traversal and fitness evaluation
- Simple implementations of crossover and mutation
- Native binary (fast) search when ordered
- Visual structure similar to a function expression

Mutation

The mutation functionality in our EC System is implemented by simply selecting a random index in an expression tree and flipping the node at that index to another corresponding node. This might mean division changing to multiplication or the constant 4 changing to 5.

Crossover

The crossover functionality in our application is implemented by selecting a random tree height, taking a node from two trees at that height, and swapping them with each other. We implement crossover in this way to prevent our expression trees from growing to unmanageable heights. This improves performance and reduces memory overhead. Other implementations might ignore this and run just fine but they do not align with our team goals regarding performance.

Evolution

Our EC System implements evolution between generations in a very straight-forward way. First, we order the population by fitness using a Comparator (see below). We take the highest two fitness expression trees and carry them into the next generation by default. Next, we take those two expression trees, deep clone them (see below), and crossover their clones. These crossover clones are then brought into the next generation. The remaining individuals in the population are made up of an even number of clones of the two crossover expression trees mutated and random expression trees. As a result of this approach, the minimum population size is 7 (the 2 highest fitness expression trees, 2 crossover expression trees, 2 crossover mutated expression trees, and 1 random expression tree). This approach is simplistic in nature and focuses on evolving very quickly. This approach, however, introduces the possibility for our system to evolve too quickly and get “stuck” going down an unsuccessful evolutionary path. We get around this limitation by regenerating our entire population after 1000 iterations of failing to reach the target fitness value.

Deep Cloning

We needed to implement deep cloning to achieve crossovers and mutations without altering the source of said operations. We achieved cloning functionality by manually iterating through a tree's properties and cloning them from one instance to another in a recursive manner. As a testing procedure we checked that mutable instances are cloned and that all properties were cloned properly. The interface `ICloneable` was used by our `ExpressionTree`.

Deep Cloning Advantages:

- Control of what will be performed
- Quick execution
- Cons of using cloning frameworks or reflection:
 - Less control of what is performed
 - Bug prone with mutable objects if the reflection tool does not clone sub objects properly
- Slower execution
- Every mutable instance is fully cloned, even at the end of the hierarchy

Interface vs. Abstract Class

The interface `IExpressionTreeNode` defines all of the methods that are required for expression trees to evaluate, mutate, etc. Both branch and non-branch nodes need to implement this interface. We chose to use an interface for this purpose over abstract classes because:

- Interfaces are more flexible, because a class can implement multiple interfaces.
- Java does not support multiple inheritance - Using abstract classes prevents our users from using any other class hierarchy. Since some of our classes do need to implement multiple interfaces this approach worked out well.

Factory Pattern

Factory classes are often implemented because they allow the project to follow the SOLID principles more closely. In particular, the interface segregation and dependency inversion principles. Factory methods define an interface for creating an object but let the subclasses decide which class to instantiate. The factory method lets a class defer instantiation to subclasses.

Factories and interfaces also allow for long term flexibility and a more decoupled solution, and therefore more testable, design. One might use this approach because:

- It allows the easy introduction of an IoC container
- It makes mocking interfaces for code testing much simpler
- It increases flexibility when it comes time to change or refactor the codebase (i.e. you can create new implementations without changing the dependent code)
- Using an IoC container to resolve your dependencies allows you even more flexibility. There is no need to update each call to a particular constructor whenever dependencies of the class are changed.
- Following the interface segregation principle and the dependency inversion principle allows us to build highly flexible, decoupled applications.

Comparator vs. Comparable

A class should implement the `Comparable` interface if that is the clear, natural way to sort instances of the class. If, however, sorting only makes sense for a specific use case, then a `Comparator` is a better option. In our case we needed to sort by the fitness values of a population of `ExpressionTree` instances to determine the most fit individuals so this approach works well.

Enums

Enums are lists of constants. When we need a predefined list of values which do not represent some kind of variable numeric or textual data, we should use an enum. In our project, we needed a predefined list of values from Zero to Nine, so we used an enum. It is easier to add values to enum class in the future. Another example of an enum in our code was a one for the different available operators. This enum could then be expanded to include methods for

executing the different operators, so that if an additional operator is introduced a method for executing it would be required. This functionality is rather unique to Java as most programming languages do not allow this.

Custom Exception Classes

We create new custom exceptions if they can relay useful information during the event of an exception. There was none in our case since we could use the existing `InvalidStateException` and `InvalidArgumentException` classes by importing from `sun.plugin`.

Final Classes

We made use of the `final` keyword on classes so that certain classes can't be subclassed (ex: `ExpressionTreeFactory`). Doing this confers security and efficiency benefits, and shows a certain level of intent as well as a developer.

Configuration

Our project team determined a few configuration requirements throughout the course of these projects. These include providing inputs for the user to to adjust the way the EC System runs. Some of these inputs and are reasoning for their default values are:

Population Size

Our EC System has a default population size of 10. Because of the aggressive nature in which our system evolves it is not beneficial to have a very large population size as unfit individuals are dropped almost immediately. Through testing, our team determined that a population size of up to 20 can be beneficial depending on the target solution. Anything greater will hinder performance. Finally, a population size of 7 is minimum (see evolution section above).

Expression Tree Height

Our application has a maximum expression tree height of 5. This limit is used mainly to help improve the performance of our system as a whole. This limit takes into consideration the target solutions we will potentially be identifying. A larger solution may not fit into an expression tree with a height of 5, which would in term be "unsolvable". This limit could be reevaluated in the scenario of larger target functions.

Target Fitness Value

Because of the way our system evolves and resets its populations we can safely target a fitness value of 0. A higher fitness value greatly improves the performance of the solution, however it was a project team goal to target perfect solutions.

Max Run Time

Our EC System defaults to the max run time of 15 minutes. This was the performance requirement given to project teams initially. Because our system usually completed in a minute or less it doesn't really matter what this is set to initially, but we did provide a minimum value of one minute to be safe.

System Testing

Whitebox Testing

Unit Testing:

As a project team, one of our goals was to have as high of code coverage (see below) with our unit tests as possible. The reason for this is that high unit testing code coverage improves the maintainable of its associated code, allows for the accurate detection of bugs, and allows for code that is easy to regression test. Our actual code coverage numbers were calculated using EclEmma. Our unit tests include several test definitions - some of the major ones include:

generationTest()

This test outputs the number of populations it takes to solve a solution and its runtime. We can also check how many times the system had to reset the population.

```
Resetting population...
Resetting population...
Resetting population...
Resetting population...
Resetting population...
Resetting population...
Took 864 population to solve: ((0 + ((1 ÷ 4) + 1)) • ((x • x) - (1 ÷ 1)))
Generation Test 1
*****
Test length: 563 ms
```

trainingDataInputOutputTest()

This test checks that all of the training data is read correctly and prints out the results.

testGetClonedExpressionTree()

This test checks if an expression tree is cloned correctly.

testCompareEquals() and testCompareNotEquals()

This test tests whether two trees are equal or not.

testFitnessValue()

This test checks whether or not an expression tree's fitness value is evaluated correctly.

Code Coverage:

To calculate our application's unit testing code coverage we ran test classes in IntelliJ alongside EclEmma. EclEmma calculates statement coverage, method coverage, and class coverage, all of which are defined by EclEmma as:

Statement Coverage

For all class files that have been compiled with debug information, coverage information for individual lines can be calculated. A source line is considered executed when at least one instruction that is assigned to this line has been executed. Due to the fact that a single line typically compiles to multiple byte code instructions the source code highlighting shows three different statuses for each line containing source code:

No coverage: No instruction in the line has been executed (red background)

Partial coverage: Only a part of the instruction in the line have been executed (yellow background)

Full coverage: All instructions in the line have been executed (green background)

Depending on source formatting a single line of a source code may refer to multiple methods or multiple classes. Therefore, the line count of methods cannot be simply added to obtain the total number for the containing class. The same holds true for the lines of multiple classes within a single source file. JaCoCo calculates line coverage for classes and source file based on the actual source lines covered.

Method Coverage

Each non-abstract method contains at least one instruction. A method is considered as executed when at least one instruction has been executed. As JaCoCo works on byte code level also constructors and static initializers are counted as methods. Some of these methods may not have a direct correspondence in Java source code, like implicit and thus generated default constructors or initializers for constants.

Class Coverage

A class is considered as executed when at least one of its methods has been executed. Note that JaCoCo considers constructors as well as static initializers as methods. As Java interface types may contain static initializers such interfaces are also considered as executable classes.

Code Coverage –

Summary

Our project achieved a satisfactory >80% total code coverage including the coverages defined above (see image on right).

Class	Class%	Method%	Line%
Settings.java	100%	93.8%(15/16)	96.6%(28/29)
Total	100%	96%(72/75)	82.1%(183/223)
ExpressionTree	100%	100%(12/12)	88.9% (32/36)
ExpressionTreeFactory	100%	90%(9/10)	79.4% (50/63)
ExpressionTreeFitnessComparator	100%	100%(2/2)	100%(3/3)
ExpressionTreeFitnessTrainer	100%	90%(9/10)	87.5%(7/8)
ExpressionTreeNodeBranch	100%	100%(11/11)	68.4%(26/38)
ExpressionTreeNodeBranchOperands	100%	100%(13/13)	100%(19/19)
ExpressionTreeNodeLeaf	100%	100%(10/10)	100%(16/16)
ExpressionTreeNodeLeafIntegers	100%	100%(5/5)	100%(17/17)
ExpressionTreeNodeVariable	100%	100%(10/10)	56.5%(13/23)
ProgramGui	100%	93%(26/28)	94.5%(263/277)

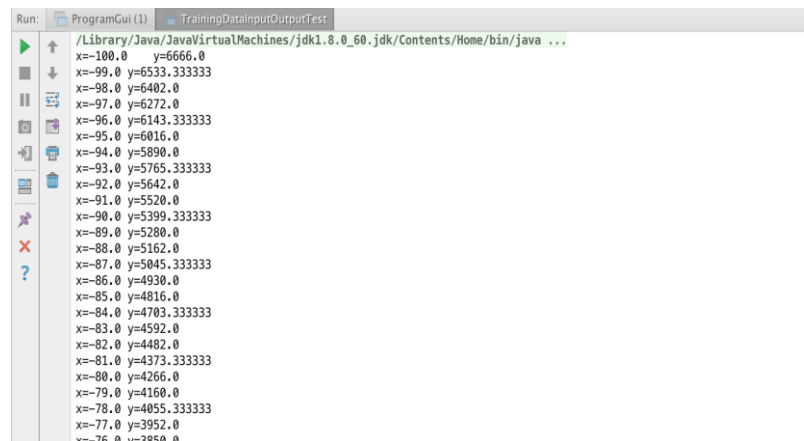
Blackbox Testing:

Our project team identified a few black box tests that would be beneficial to run. These tests are less programmatic in nature and would typically be run by someone in QA.

Data Input / Output Test

The Data I/O Test (DIOT) is designed to test the I/O component of the Evolutionary Computing Application. The Evolutionary Computing Application operates after the user supplies input training set data. The training data file consists of x,y pairs separated by semicolons.

First the test reads in the supplied training data file and creates a DataSet object within the application. The TrainingDataSet contains a list of TrainingData objects that correspond to each line of the input file. Each TrainingData contains the x,y values for that input. After building the training data objects, the test outputs the contents of the TrainingDataSet object to the system console:



```
Run: ProgramGui (1) TrainingDataInputOutputTest
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/bin/java ...
x=-100.0 y=6666.0
x=-99.0 y=6533.333333
x=-98.0 y=6402.0
x=-97.0 y=6272.0
x=-96.0 y=6143.333333
x=-95.0 y=6016.0
x=-94.0 y=5890.0
x=-93.0 y=5765.333333
x=-92.0 y=5642.0
x=-91.0 y=5520.0
x=-90.0 y=5399.333333
x=-89.0 y=5280.0
x=-88.0 y=5162.0
x=-87.0 y=5045.333333
x=-86.0 y=4930.0
x=-85.0 y=4816.0
x=-84.0 y=4703.333333
x=-83.0 y=4593.333333
x=-82.0 y=4482.0
x=-81.0 y=4373.333333
x=-80.0 y=4266.0
x=-79.0 y=4160.0
x=-78.0 y=4055.333333
x=-77.0 y=3952.0
x=-76.0 y=3849.0
```

This test is helpful in verifying that the program is receiving the input exactly as it should be.

Data Structure Test

The Data Structure Test (DST) is designed to test the underlying data structure of the Evolutionary Computing Application. This test runs the EC System iteratively in order to determine that the system is generating and maintaining expression trees properly. The solution found, the total number of populations, and the total runtime are output. These results can then be analyzed by QA. The image below shows an example of the test output:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/bin/java ...
Took 289 population to solve: (((x ÷ 4) • x) - (5 ÷ 4)) + (((6 + x) - 6) • x))
Generation Test 1
*****
Test length: 47 ms
```

Metrics

Our project team decided that using the third-party library Metrics Reloaded would be the most efficient way to calculate our application's metrics. While the calculations themselves are simple, the information and analysis of their meaning is more complex -- and, very important. Software metrics

provide numerical evidence to help reinforce design decisions. They also provide a measure of the software complexity, ease of maintenance, and any structural design inefficiencies which could be improved.

Complexity Metrics

Cyclomatic Complexity

Cyclomatic complexity is the quantitative measurement of the number of linearly independent paths in our classes. We computed the average operation cyclomatic complexity by class:

Class	Avg Operation Cyclomatic Complexity
gui.ProgramGui	1.8
settings.Settings	1
tree.ExpressionTree	1.33
tree.ExpressionTreeFactory	2.25
tree.ExpressionTreeFitnessComparator	1
tree.ExpressionTreeFitnessTrainer	3
tree.ExpressionTreeNodeBranch	1.8
tree.ExpressionTreeNodeBranchOperands	1
tree.ExpressionTreeNodeLeaf	1.2
tree.ExpressionTreeNodeLeafIntegers	1
tree.ExpressionTreeNodeVariable	1.2

Looking at these numbers we can determine that the highest average cyclomatic complexity is held by the tree.ExpressionTreeFitnessTrainer class. Seeing as how this number is well below 10, we can conclude that our classes exhibit a high level of cohesion, or in other words the operations in our classes are not trying to do more than they should be (SOLID single responsibility principle).

Weighted Method Complexity

The weighted method complexity metric is the sum of the complexities of all of a class' methods. It is an indicator of how much effort is required to develop and maintain a particular class. Our results are as follows:

Class	Weighted Method Complexity
gui.ProgramGui	27
settings.Settings	14
tree.ExpressionTree	16
tree.ExpressionTreeFactory	18
tree.ExpressionTreeFitnessComparator	1
tree.ExpressionTreeFitnessTrainer	3
tree.ExpressionTreeNodeBranch	18
tree.ExpressionTreeNodeBranchOperands	7
tree.ExpressionTreeNodeLeaf	12
tree.ExpressionTreeNodeLeafIntegers	3
tree.ExpressionTreeNodeVariable	12

Our gui.ProgramGui has the highest weighted method complexity meaning that it is the hardest to maintain. This makes some sense as it is the melting pot where all of the other functionality joins together. A way to possibly remedy this high number might be to introduce a new class that handles some of the execution of the EC System as opposed to delegating that work to the gui.ProgramGui class itself. Regardless, all of our numbers are well below 50, showing that our classes are quite maintainable and should be easy to develop on top of if need be.

Dependency Metrics

Cyclic Dependencies

A cyclic dependency is a dependency in which two or more classes directly or indirectly rely on each other to function properly. Cyclic dependencies can be a symptom of improper callbacks, odd coupling, or poorly written observer patterns. Our application has 0 cyclic dependencies.

Dependencies

A dependency is a class that another class depends on to function properly. High counts of dependencies can be symptoms of high coupling in an application, ultimately inferring lower code maintainability and reusability. Our dependency counts are:

Class	Dependencies	Transitive Dependencies
gui.ProgramGui	5	5
settings.Settings	0	0
tree.ExpressionTree	1	1
tree.ExpressionTreeFactory	1	1
tree.ExpressionTreeFitnessComparator	0	0
tree.ExpressionTreeFitnessTrainer	1	1
tree.ExpressionTreeNodeBranch	0	0
tree.ExpressionTreeNodeBranchOperands	1	1
tree.ExpressionTreeNodeLeaf	0	0
tree.ExpressionTreeNodeLeafIntegers	1	1
tree.ExpressionTreeNodeVariable	0	0
tree.ICloneable	0	0
tree.IExpressionTreeNode	0	0

As shown, our gui.ProgramGui class has the highest dependency count. Again, this is likely because it is where a lot of the major system execution occurs. And again, a remedy for this would be delegating the system execution work to one or more worker classes. Otherwise, our other classes have 1 or 0 dependencies, suggesting that our code has relatively low coupling.

Dependents

A dependent is a class that is depended on by other classes. If too many classes depend on a single class, it might mean that making modifications to the dependency would be very difficult, i.e. a sign of high coupling. Our dependent numbers are:

Class	Dependents	Transitive Dependents
gui.ProgramGui	0	0
settings.Settings	7	12
tree.ExpressionTree	5	5
tree.ExpressionTreeFactory	3	3
tree.ExpressionTreeFitnessComparator	3	3
tree.ExpressionTreeFitnessTrainer	3	3
tree.ExpressionTreeNodeBranch	0	0
tree.ExpressionTreeNodeBranchOperands	0	0
tree.ExpressionTreeNodeLeaf	3	3
tree.ExpressionTreeNodeLeafIntegers	3	3
tree.ExpressionTreeNodeVariable	0	0
tree.ICloneable	0	0
tree.IExpressionTreeNode	0	0

Our class that is depended on the most is the settings.Settings class. That is fine as the settings.Settings class is a singleton and is intended to be used as such. Our next highest dependent count is our expression tree with 5, which happens to be a major class in our application. Overall, these numbers still indicate relatively low coupling among classes.

Lines of Code Metrics

Lines of Code

This metric is straightforward - it simply tells us the lines of code in each module:

Module	Lines of Code
main.java.gui	361
main.java.settings	53
main.java.tree	490
main.java.util	69

This metric can be useful when trying to refactor for simplification purposes and identifying whether or not the number of lines of code is going down. It is not a real indicator on the code quality alone.

Cohesion Metrics

Lack of Cohesion

Cohesion is the idea that a class should represent a single abstraction. A low cohesion suggests that a class should be broken out into multiple sub-classes. The lack of cohesion is a numerical inverse of the level of cohesion in a class. The measurement is typically done in this way so that “lower values are better”. Our lack of cohesion numbers are:

Class	Lack of Cohesion
gui.ProgramGui	2
settings.Settings	7
tree.ExpressionTree	1
tree.ExpressionTreeFactory	4
tree.ExpressionTreeFitnessComparator	1
tree.ExpressionTreeFitnessTrainer	1
tree.ExpressionTreeNodeBranch	1
tree.ExpressionTreeNodeBranchOperands	3
tree.ExpressionTreeNodeLeaf	6
tree.ExpressionTreeNodeLeafIntegers	2
tree.ExpressionTreeNodeVariable	9

As shown above some of our classes exhibit high lack of cohesion numbers. This might indicate that these classes are not following the single responsibility principle closely enough. Taking `tree.ExpressionTreeNodeVariable`, for example, we see that the class has several functions to help assist in the recursive operations of an expression tree. These functions all are necessary but are creating a false positive in our lack of cohesion metrics. Other classes, such as `settings.Settings` and `tree.ExpressionTreeNodeLeaf` exhibit these same false positives. After removing the outliers, we are left with a relatively cohesive set of classes. This falls more in line

with what the results of our other metrics are indicating, a relatively cohesive, maintainable codebase.

Project Results

As a whole our project was successful. Our evolutionary computing system finds solutions for a given set of inputs (with a fitness of 0 for the training sets of the two defined functions during the project), and adheres to both the project's initial set of requirements and the requirements our team further defined. More importantly, however, our project includes metrics and other quantitative results as proof of a structured, maintainable, and accountable code base, including: unit test results that all pass, a total code coverage of more than 80%, code metrics that indicate a relatively low number of lines of code, complexity, and dependencies, other testing functions that all pass, and an issue tracking system that shows that all outstanding issues and tasks have been completed or have been accounted for.

Bibliography

Software Engineering: A Practitioner's Approach, by Roger Pressman, Bruce Maxim, McGraw-Hill, 2014.

Chih Lai, Ph.D. SIES 610 Software Engineering, *Lecture Slides*. St. Thomas University. Summer 2016.

Eclipse.com, Eclipse Logo. <https://eclipse.org/eclipse.org-common/themes/solstice/public/images/logo/eclipse-800x188.png>, July 2016.

Google.com, Google Logo. <http://dwglogo.com/wp-content/uploads/2016/06/Google-2015-logo.png>, July 2016.

Gitlab.com, Gitlab Logo. <https://gitlab.com/uploads/project/avatar/13083/gitlab-logo-square.png>, July 2016.

Osx.vn, IntelliJ Logo. <http://osx.vn/attachments/ij-png.4895>, July 2016.

Google. Google Docs. 2016. [Cloud Storage] <https://drive.google.com/drive/folders/xxxxxxxxx>.

<https://www.quora.com/In-Java-when-should-you-use-an-interface-instead-of-an-abstract-class>

<http://programmers.stackexchange.com/questions/253254/why-should-i-use-a-factory-class-instead-of-direct-object-construction>



Appendices:

Appendix A: Software User Manual

About This Guide

This guide is intended for anyone who wants to use this Evolutionary Programming System.

System Requirements

In order to execute the program, Java Runtime Environment (JRE) version 7 is needed. If you need JRE 7 to be installed on your machine to allow you to run this program, you can go to Sun.com and search for JRE 7 or go to the following link to download a copy of JRE 7:

<http://java.sun.com/javase/downloads/index.jsp>.

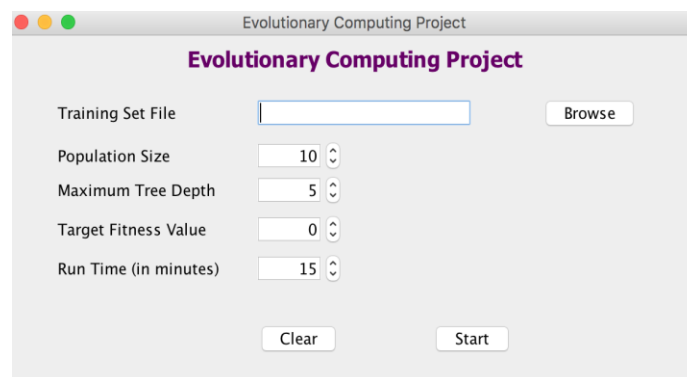
****In addition, you will need a command prompt if you would like to view the program output.*

Executing the Evolutionary Computing Application

1. Go to EC_System through command line using commands:
 - a. `cd Downloads/EC_System` (Assuming it's downloaded/located in Downloads Folder)
2. Install Maven (if you don't already have it in your machine):
 - a. Mac: Brew install mvn
 - b. Windows: Refer to <https://www.mkyong.com/maven/how-to-install-maven-in-windows/>
3. Run "`mvn exec:java`".

```
[INFO] ~~/Downloads/EC_System
[23:31 $ mvn exec:java
/Users/z001ttz/.mavenrc: line 1: unexpected EOF while looking for matching `"'
/Users/z001ttz/.mavenrc: line 3: syntax error: unexpected end of file
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building ece-maven 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) > validate @ ece-maven >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) < validate @ ece-maven <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ ece-maven ---
```

After executing the file, the GUI will appear on your screen as shown below:

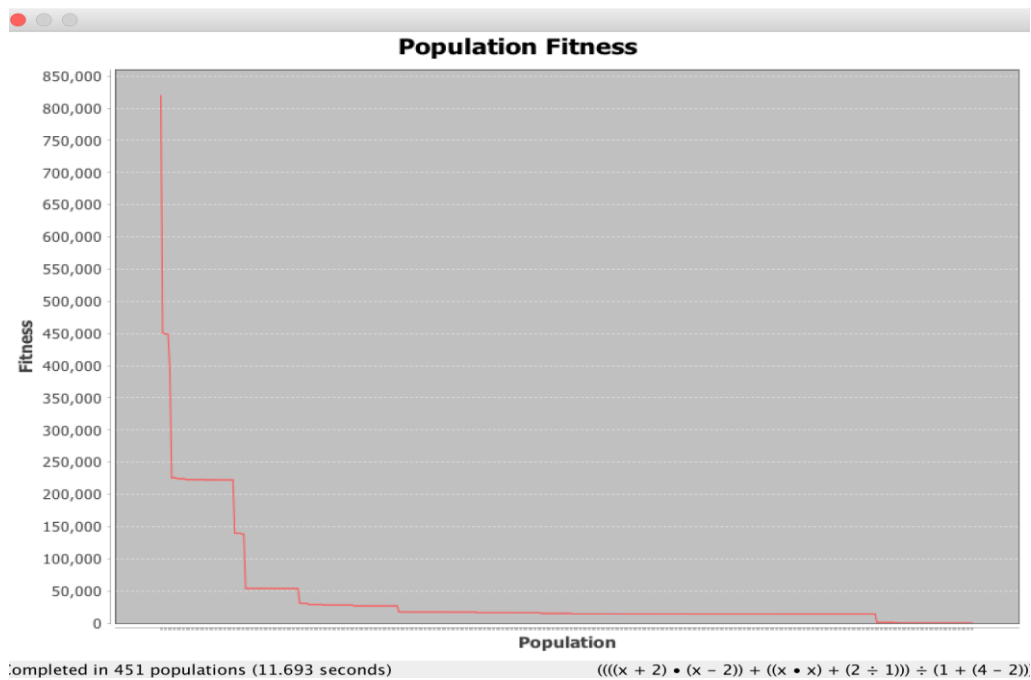


4. Populate the fields and the location of the training set file.

The screenshot shows a window titled "Evolutionary Computing Project". Inside, there's a section titled "Evolutionary Computing Project" in purple. Below this, there are several input fields and buttons:

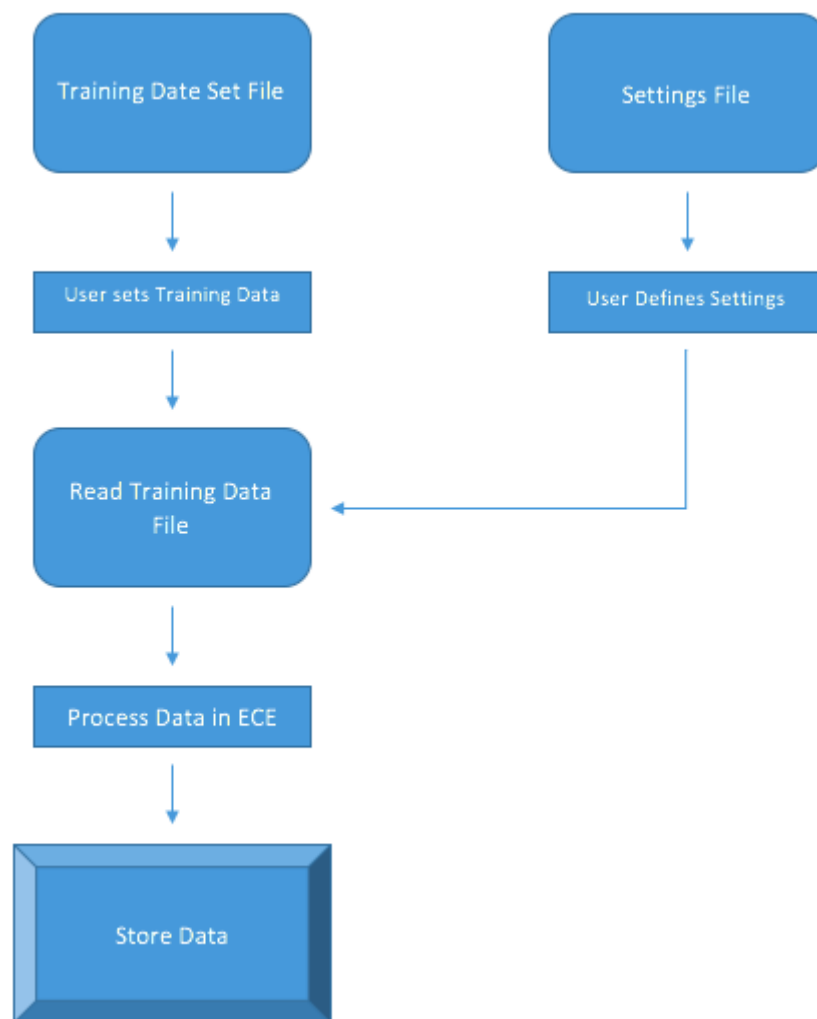
- Training Set File:** A text box containing "/EC_System/training-set.txt" and a "Browse" button.
- Population Size:** A spin box set to "10".
- Maximum Tree Depth:** A spin box set to "5".
- Target Fitness Value:** A spin box set to "0".
- Run Time (in minutes):** A spin box set to "15".
- At the bottom, there are "Clear" and "Start" buttons.

After you click start, it will execute the application and generate the results with a graph of population vs fitness.

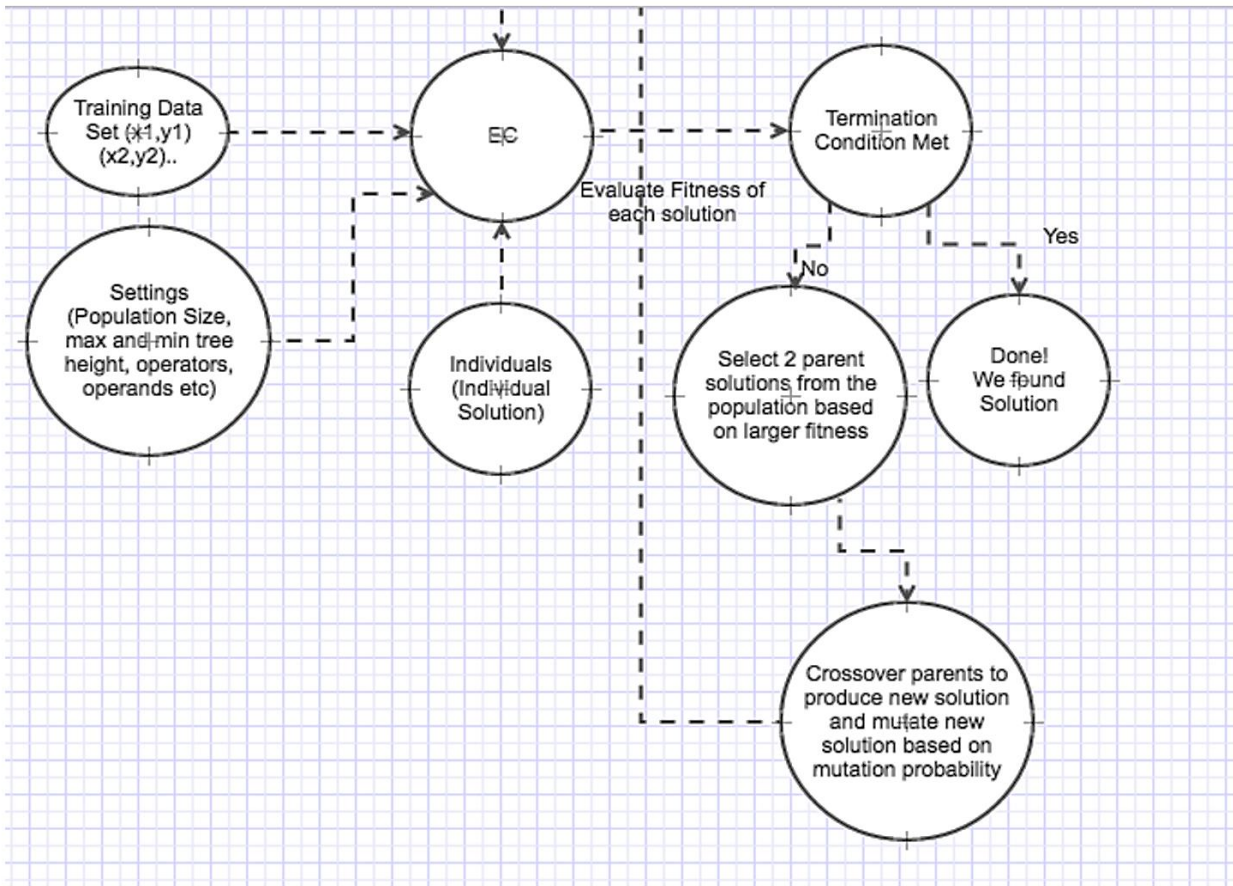


Appendix B: Diagrams & Visuals

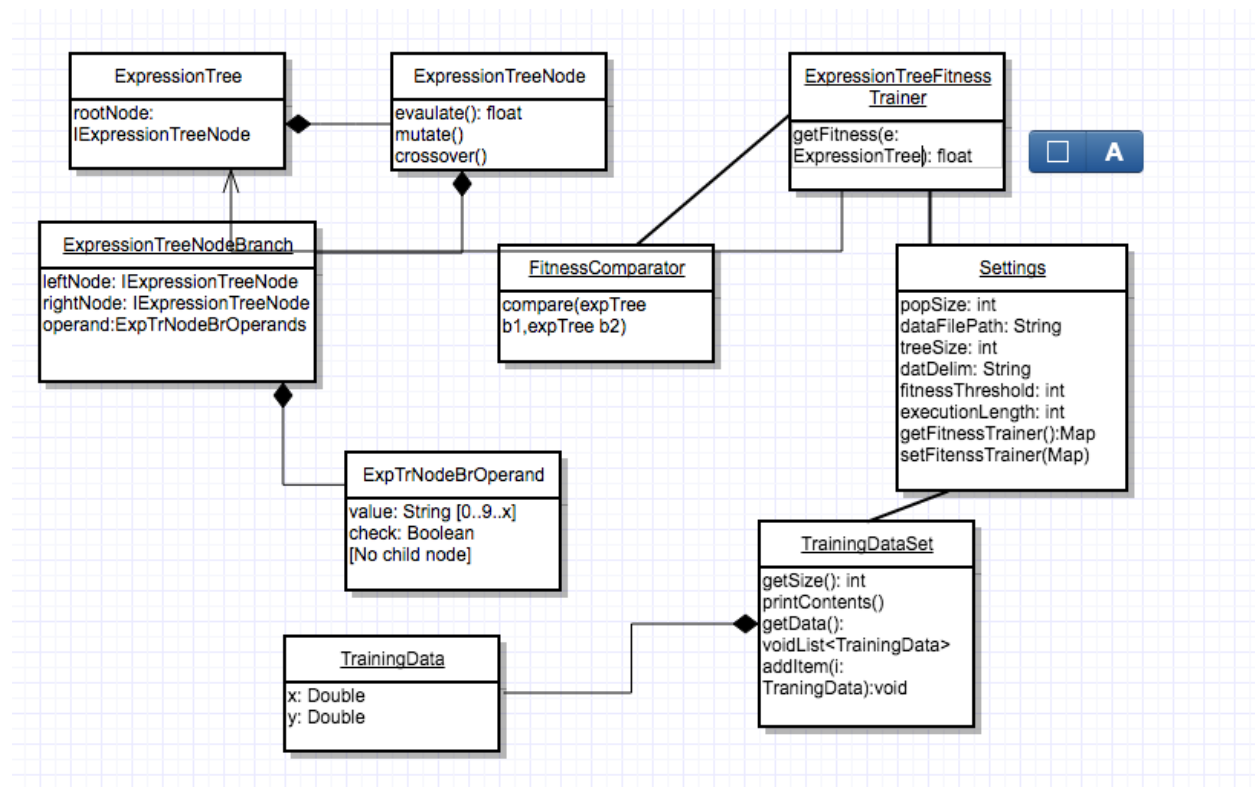
Dataflow Diagram:



Finite State Machine:



Object Oriented Model:



Appendix C: Post-Project Analysis

Lessons Learned:

- The GitLab issue tracking tool should have been used to track more non-development related tasks.
- There are several ways to test, measure, assess software code and structure, but the metrics of most interest are dependent upon the software requirements, objectives, and specific application
- Software development (engineering) requires a significant amount of outlining, (re)structuring, and upfront planning to ensure a successful result...writing code was the easy part
- Better group assignments and accountability to prevent overlapping/gaps in key it4ems
- Better insight into project status and what work has been done by each group member -> conflicting schedules and full time jobs made it difficult to stay on “same page”
- Our project team should have better defined “management” roles. If a group manager was specifically assigned, the two items above may have also been solved.

Future Work

- Enhance GUI functionality
 - Ex: Add feature that allows for an automatic generation of a training set by giving user option to specify the training set input values (range) and step size
- For further testing purposes and program simplification.
- Feature request that includes porting the application to mobile.
- Java would allow easier port to Android.
- Feature enhancement to include more output options.
- Auto generate additional graphs, tables, metrics, or other visuals based on user preference

Group Collaboration Tools (Pros/Cons, Decisions)

Google Drive

Pros

- Free
- Familiar to all team members
- Real-time collaboration on Google Docs

Cons

- Not as intuitive as other drive options
- Google Docs not as familiar as MS Office to team members
- Does not integrate with Office products very well

PowerPoint

Pros

- Familiar to all team members
- Simple layout structure
- Professional looking presentations

Cons

- Not as “fancy” as other presentation options
- Not free
- Does not integrate with Google Docs very well

Prezi (considered but not used)

Pros

- Great looking presentations
- Free

Cons

- Majority of group not familiar with the tool
- Lacks linear structure like that of PowerPoint
- Not easy to learn
- Does not integrate well with Office products

GitLab

Pros

- Free
- Private repositories
- Issue tracking
- Source control and versioning
- Compatible with Git

Cons

- Not as familiar to all team members

GitHub (considered but not used)

Pros

- Free
- Issue tracking

- Source control and versioning
- Compatible with Git
- Familiar to all team members

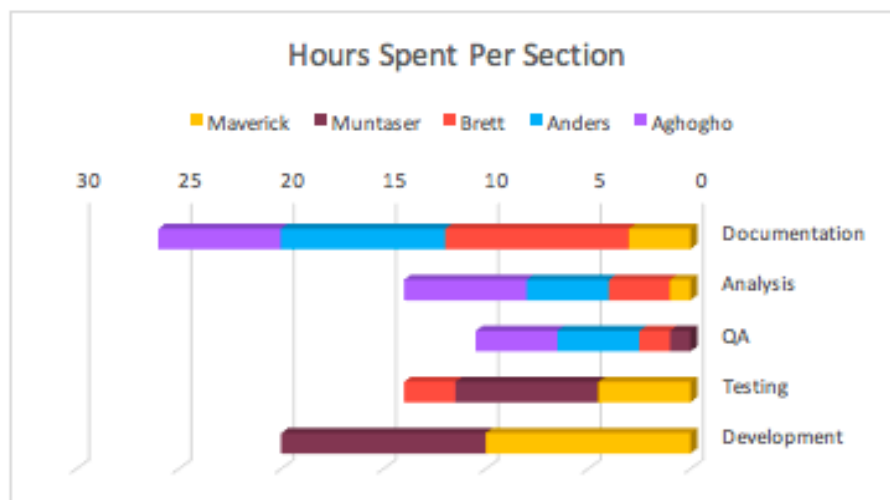
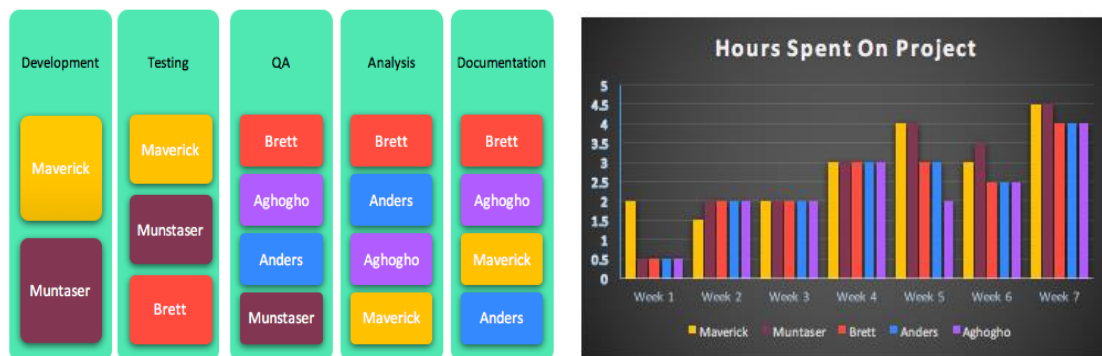
Cons

- No private repositories

Group Member Assignments and Time Allocation

We broke our team assignments into general sections: Program Development, Testing, QA (Quality Assurance), Analysis (Code, Post, Requirement), and Documentation (Presentation, Project Plan, & EC Project Paper).

As expected, our workload increased over time and was at its highest towards the end of the semester. Development, Testing and Documentation took up the most time. Development and testing would have been the most but we had to redo some presentation and documentation pieces. Areas for improvement are a better definition of member roles early on. We procrastinated a bit in terms of role delegation as well as deciding on what presentation tool we were going to use. We started using Prezi, but due to lack of knowledge of the appliance, we redid the presentation days before the due date.



Appendix D: Weekly SCM Files / Meeting Notes

Date: 6/1/2016 – Wednesday, Week 1

Time: 5:45-9:00 (3 hours 15 min)

Where: OSS333 (in class)

Attendees: Full Class

Subject: EC Project Introduction (High Level) Part 1

Summary:

- Evolutionary Computing Project was introduced by Professor Lai, outlining the project's objectives and deliverables.
 - Deliverables:
 - Project Plan - Midterm
 - Final Project and Presentation – End of Semester
 - Details found in Lecture Notes on Blackboard.
- Maximum number of individuals per group is 5

Key Action Items:

- Finalize Group members and availability (by 6/13)

****These notes were taken prior to group formation or starting any work on the team work on the project, and are for documentation and reference purposes.*

Date: 6/6/2016 – Monday, Week 2

Time: 5:45-9:00 (3 hours 15 min)

Where: OSS333 (in class)

Attendees: All

Subject: EC Project Introduction (High Level) Part 2

Meeting Summary:

- Learned additional details regarding project requirements from Professor Lai
 - Problem Definition/Validation Criteria
 - Generate a function equivalent to $(x^2 - 1) / 2$
 - Problem will change 2 weeks before final presentation
 - Must be able to adapt code accordingly
 - Constraints
 - 15 min maximum for final presentation and demonstration

Key Action Items:

- Finalize Group members and availability (due 6/13)

****These notes were taken prior to group formation or starting any work on the team work on the project, and are for documentation and reference purposes.*

Date: 6/8/2016 – Wednesday, Week 2

Time: 5:45-9:00 - Before, after, and during class break (20 min)

Where: OSS333

Attendees: All (Maverick, Muntaser, Brett, Anders, Aghogho)

Subject: Group Members Introductions, Availability, and Collaboration Platform

Meeting Summary:

- Group members finalized (5)
- All members gave introductions, including background experience, contact info, and areas of expertise
- Maverick shared his initial progress on the project with team which he had completed already individually
 - Expression Tree Classes and Methods (first iteration)
- Group Decisions
 - Programming Platform: Java
 - Software Management Platform (code): GitLab
 - Communication and Document Collaboration Platform: Google Drive

Key Action Items:

- Finalize Group members, meeting times, and assign initial project tasks based on background and areas of expertise (due 6/13)
- Create Gitlab and Google accounts (if needed) and obtain access to shared online repositories (due 6/13)

Date: 6/13/2016 – Monday, Week 3

Time: 5:45-9:00 - Before, after, and during class break (20 min)

Where: OSS333 (in class)

Attendees: All (Maverick, Muntaser, Brett, Anders, Aghogho)

Subject: Progress Check-up

Meeting Summary:

- All members are confirmed to have access to google drive and have access to the Gitlab repository
- Assigned sections of project and project plan to individual group members
- Completed first iteration of project code with basic functionality, and defined specific questions regarding project requirements and deliverables. See excerpt of email from Maverick to team for further details:

ExpressionTreeFactory

- - Can build random expression trees and return a population of trees using a preset population size. These trees have random heights using present min and max tree heights.
- - Provides an interface for working with trees, such as crossing over two trees, mutating trees, and cloning trees.
- - Crossover - works by taking two random nodes of two trees and swapping them in place
- - Mutate - works by selecting a random node of a tree and changing it depending on the node type, i.e. changing the integer value or the operator type.
- - Cloning - clones a tree such that if one is edited the other remains the same, i.e. java deep cloning

ExpressionTreeFitnessTrainer

- - Calculates the fitness of a tree given a hardcoded fitness training set and a hardcoded target function.

Further requirements needed

- - How many crossovers per generation?
- - Once the crossovers have been passed down, do we populate the rest of the population using mutations of those crossovers?
- - I.e. what happens between populations?
- - Is the target function going to be hardcoded in the final product or do we need to be able to adapt given a function string? - hopefully not
- - is the training set going to be hardcoded in the final product or do we need to be able to edit these in settings or something?
- - How close to a fitness of 0 do we need to get before determining we reached the target function and end execution?
- - Does the random tree min and max height need to be editable in settings or hardcoded to a value? - maybe doesn't matter?

Key Action Items:

- Project Plan (Due 6/22)
- Obtain answers to the questions shown above (requirements)

Date: 6/15/2016 – Wednesday, Week 3

Time: 5:45-9:00 - Before, after, and during class break (20 min)

Where: OSS333 (in class)

Attendees: All (Maverick, Muntaser, Brett, Anders, Aghogho)

Subject: Progress Check-up

Meeting Summary:

- OO Diagram for project complete and approved by all team members
- Project Plan Outline and initial documents have been uploaded to google drive

Key Action Items:

- Project Plan (Due 6/22)
- Obtain answers to the questions shown in meeting notes from 6/13 (requirements)

Date: 6/20/2016 – Wednesday, Week 4

Time: 5:45-9:00 - Before, after, and during class break (20 min)

Where: OSS333 (in class)

Attendees: All (Maverick, Muntaser, Brett, Anders, Aghogho)

Subject: Progress Check-up

Meeting Summary:

- Discussed Project Plan progress status, identified items which are not yet complete, and assigned specific tasks to group members:
 - Problem Description: Maverik
 - Requirements Analysis & Preliminary System Design: Maverik & Muntaser
 - Weekly SCM files and folders: Anders and Aghogho
 - Work Plan: Brett

Key Action Items:

- **Project Plan (Due 6/22)**
- Obtain answers to the questions shown in meeting notes from 6/13 (requirements)
- Formalize SCM Tool / format to use in second half of semester after this topic has been discussed in class next week (Week 4)

Date: 6/27/2016 – Monday, Week 5

Time: 5:45-9:00 - Before, after, and during class break (20 min)

Where: OSS333 (in class)

Attendees: All (Maverick, Muntaser, Brett, Anders, Aghogho)

Subject: Progress Check-up

Meeting Summary:

- Discussed corrections made on the returned project plan
- Input and thoughts on data flow diagram, lessons learned, visual materials and user manual
- Muntaser and Maverick shared progress on system with team
- Code metrics have been uploaded to google drive

Key Action Items:

- Project Presentation (Due 7/18)
- Prepare a user manual

Date: 7/06/2016 – Wednesday, Week 6

Time: 5:00-9:00 - Before, after, and during class break (1hr)

Where: OSS333 (in class)

Attendees: All (Maverick, Muntaser, Brett, Anders, Aghogho)

Subject: Progress Check-up

Meeting Summary:

- Evaluated system progress based on;
 - Generation of a function equivalent to $(x^2 - 1) / 2$
 - Time taken to achieve target function or function equivalent
 - Number of system resets
 - Number of generations
- Tentative Project Presentation slides have been uploaded to google drive by Brett

Key Action Items:

- Project Presentation (Due 7/18)
- Create GUI for the EC system
- Set system to re-initialize after 1,000 generations

Date: 7/13/2016 – Wednesday, Week 7

Time: 5:30-9:30 - Before, after, and during class break (1hr)

Where: OSS333 (in class)

Attendees: All (Maverick, Muntaser, Brett, Anders, Aghogho)

Subject: Progress Check-up

Meeting Summary:

- Noted changes to be made according to the new project function given by Professor Lai
 - New Problem Definition/Validation Criteria
 - Generate a function equivalent to $(2x^2 - 2) / 3$
- Discussed testing methods to be implemented

Key Action Items:

- **Project Presentation (Due 7/18)**
- Upload completed project documents and slides to google drive

Date: 7/14/2016 – Thursday, Week 7

Time: 5:45-8:00

Where: OSS326

Attendees: All (Maverick, Muntaser, Brett, Anders, Aghogho)

Subject: Progress Check-up

Meeting Summary:

- Discussed project status, identified uncompleted sections of project and final report, and assigned specific tasks to group members:
 - System changes & Testing implementation: Maverik & Muntaser
 - User manual: Anders
 - Visual Materials: Brett
 - SCM files and folders: Aghogho

Key Action Items:

- **Project Presentation (Due 7/18)**
- Complete assigned tasks