FLATLAND – ABBOTT'S FLATLAND USING ARTIFICIAL LIFE*

PROJECT REPORT SUBMITTED TO THE IT UNIVERSITY OF COPEHAGEN IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR IN SOFTWARE DEVELOPMENT

Anders Høst Kjærgaard**

2012

*Supervised by Professor Joseph Roland Kiniry, Department of Informatics and Mathematical Modelling, Technical University of Denmark, 2800 Kgs. Lyngby

**ahkj@itu.dk, Software Development Student, IT University of Copenhagen, 2300 Copenhagen S.

Abstract

This report describes a system, Flatland, that uses genetic programming to evolve the behavior of agents. The system was developed by the author at ITU during the summer of 2012 as a bachelor's project. Flatland is currently hosted as an open-source project at http://flatland.codeplex.com

Flatland seeks to be a realization of Edwin Abbott's novella: *Flatland: A Romance of Many Dimensions.* and suggest how Flatland can be brought to life using ALife. A study of genetic programming founded the basis for creating the solution, and this report examines how the result was achieved.

The result of the work is the Flatland software, where the life of the Flatlanders will evolve based on what genotypes are made available to them. The core of the implementation is its GP implementation, which can be used as a general search engine, but in its place seeks to join the idea of searching for a problem with teaching an agent to behave in a given way in its environment.

This report describes how I developed the software through analyzing different theories and combining the findings into a novel solution. It is my hope that this work will catch interest from others and continue to evolve in the future.

Acknowledgements

I would like to thank Joseph Roland Kiniry (ITU, soon DTU) for supervising this project and giving me the chance to undertake its realization. It is not the first time he has given me the possibility to work on an inspiring project, and I am very grateful for that – thanks Joe.

A thanks also goes to Michael O’Neill (UCD) and Simon Bang Therkelsen (ITU), both for pointing me towards great material that became essential to how this project took form.

Finally a thanks to my girlfriend Lea for her support during this period.

Contents

Glossary and List of Abbreviations

AI – Artificial intelligence
Crossover – Method for exchanging two sub trees.
EA – Evolutionary algorithm
Farseer – Farseer physics engine
Full initialization – Building a tree with equal depth at every leaf
Function – A member of the function set in GP
Terminal – A member of the Terminal set in GP
GA – Genetic algorithm. Belongs to the EAs
GP – Genetic programming
GP techniques– Initialization, Crossover, Mutation, Selection
Genotype – An organism's genetic blueprint.
Grow initialization – Building a tree with possible variable depth at leaves
Mutation – Method for altering a tree
Phenotype – The actual appearance of the genotype after influence of the environment.
Ramped half and half – half grow, half full initialization
Selection – Choosing which individuals of a population that survives to the next generation
XNA – Microsoft's XNA Game Studio

1 Problem

In this section some background and motivation for the project is stated. In section 1.2 the problem and type of system we want to be able to build is presented, and in section 1.3 a corner of the field is analyzed to understand how this can be achieved.

1.1 Background and motivation

Artificial Life (ALife) is a broad field of study that deals with examining living systems in artificial environments. The field has evolved to have practical importance and has become the study of research for many IT-related areas. But it also raises questions of existential character, like: what is life, or, what is intelligence? The use of artificial intelligence (AI) in modern computing has become so pervasive that we don't even think about its presence anymore. Computers that can predict our intentions or replicate human behavior, e.g. in games and interactive learning software, have merely become the de facto standard of what computers are capable of. In many ways, the line between artificial life and real life is fading away and we are beginning to realize, that the frontier is only bounded by our comprehension, and not by technology. Today, modern science is capable of producing living cells from the non-living[1], and according to Danish scientist Steen Rasmussen, living technologies will not only change the way we solve problems, but the whole way we organize and understand society[2]. The study of living systems and AI is not only an interesting field due to its immediate practical appliances, but indeed also for its existential implications.

In biology, the interest for artificial life is in particular due to the one circumstance: “Evolution is extremely slow”[3]. Computer systems like Avida[4] aims at creating computer environments that replicate real world environments accurately enough to be studied as if they were the same thing. Still, this can be very difficult or even impossible with current technology. Central to the field is the theory of evolution, concretized in computing by the evolutionary algorithm (EA). The behavior of a population is formed by continuously applying the notion of natural selection, breeding and mutation, and by doing so, evolve a solution to how life can take its form in a complex environment. Many subsets of the EA exist, and it has been used to solve wide range of problems. A particular characteristic is its ability to, at times, come up with highly optimal solutions to problems where other search algorithms might get stuck in a poor local optimum. Among some of the more extraordinary results has been the use of genetic programming (GP), to duplicate existing patented inventions, and even invent one.[5]

[1] http://politiken.dk/videnskab/ECE976352/forskere-skaber-kunstigt-liv/
[2] http://www.youtube.com/watch?v=0cZQRJZPXt4
[3] http://avida.devosoft.org/about/
[4] http://avida.devosoft.org/
[5] [1] page 119

Central to working with ALife in software is teaching computer programs to act in a human-like or human-competitive way, and in that sense life is encompassed by what intelligence is. Koza[6] lists 8 different criteria that an automated computer result can satisfy to be considered “human-competitive”.[7] This intelligence and ability to make machines learn (Machine-learning, ML) has become a method that is applied in nearly every area of the IT industry, and it is essential that today's software developer has knowledge and understanding of the field. Therefore the motivation behind this project is a mixture of pure curiosity to how these methods work, and to get firsthand experience with the field.

1.2 Problem description

Given the descriptions of the shapes and characters in Flatland, we can model a virtual world, using Farseer Physics Engine (FPE) and Microsoft XNA Game Studio (XNA), which we can interact with from our real 3-D as it is done in the book. Our main problem statement reads:

How can Flatland be realized with artificial life?

Ultimately, to address this problem, we want to build a system that can simulate the artificial life of Flatland. Based on a study of a known machine-learning technique, we can implement a method that can evolve the behavior of the creatures - the ability to be able to do so is central to the problem. Specifically, this can be achieved by implementing the genetic programming (GP) techniques described in *A Field Guide to Genetic Programming*[8] and by studying existing implementations.

To address the problem in a broader sense, the system should allow for some user interaction and feedback about what is going on in the environment. To actually *realize* Flatland, the final product should both have some visual quality and be extensible for future development. Giving Flatland artificial life is also a question of making it a system that others may contribute to give life to.

1.3 Problem analysis

The central part of the problem stated above, is to be able to implement a technique that can make the creatures, or agents, learn. A widely used method is that of GP, which by recommendation was chosen for this project[9]. However, the field offers many varieties of evolutionary computation that could have been used. Neural networks could have been an alternative, or the grammatical evolution (GE) and an open-source system Geva[10] that was considered, but laid aside as it would lead to an implementation where Flatland would make use of an existing system for the ML component - which would be against the

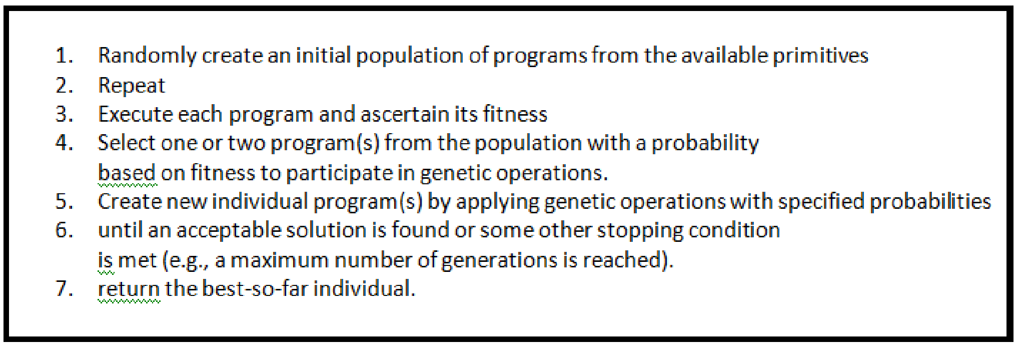
6 John Koza, Standford, pioneer in the field of genetic programming, owner of Genetic programming Inc. http://www.genetic-programming.com/
7 [Poli, Langdon, McPhee; A Field Guide to Genetic Programming][1] page 118
8 http://www.gp-field-guide.org.uk/
9 See acknowledgements
10 http://ncra.ucd.ie/Site/GEVA.html and http://ncra.ucd.ie/geva/geva.pdf

motivation behind this project. Due to a tight time-table, this project has mostly focused on the GP methodology; however, a few things in GEVA have been used as inspiration.
For the rest of this section we will analyze different available techniques that can help us solve the problem at hand.

1.3.1

Genetic Programming

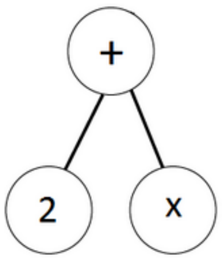
Genetic programming is a machine-learning technique based on the genetic algorithm (GA) that automatically produces programs that can solve a given problem. The only thing we need to worry about is to define the problem at a high level, and not how the solution is actually formed. What distinguishes GP from other techniques is, that it searches through different compositions of program code until it finds a program that satisfies a given problem. The algorithm is:



```
1.  Randomly create an initial population of programs from the available primitives
2.  Repeat
3.  Execute each program and ascertain its fitness
4.  Select one or two program(s) from the population with a probability
    based on fitness to participate in genetic operations.
5.  Create new individual program(s) by applying genetic operations with specified probabilities
6.  until an acceptable solution is found or some other stopping condition
    is met (e.g., a maximum number of generations is reached).
7.  return the best-so-far individual.
```

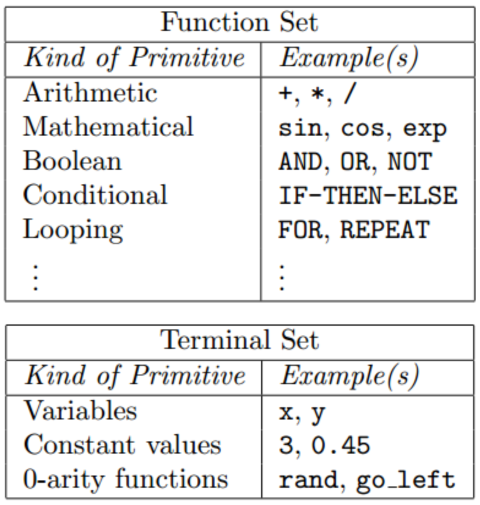
Algorithm 1, as defined in Field Guide To Gennetic Programming [1]

The most common form of GP is to represent programs as syntax trees. Writing in prefix notation, figure 1 represents the arithmetic expression + 2 x.



Figur 1 Syntax tree representing 2 + x

The expression in figure 1 represents a program with a primitive set consisting of ‘2’, ‘x’ and ‘+’. These primitives are the building blocks that can appear in the program. In GP we divide the primitives into a function set and a terminal set. Primitives of the function set are those that can be placed at internal nodes, and primitives of the terminal set are those that can be placed at the leaf nodes. As such, to be able to define what a program can consist of we need to provide the primitive set. In general, the terminal set may consist of variables, zero-arity functions and constants. The members of the function set is relative to the problem we are trying to solve, in figure 1 this could be arithmetic operations '+', '*', '-' etc. These sets, however, can contain any given types of functions and terminals relative to the problem domain. See figure 2 for examples. Some primitives may need to be *protected*, e.g. division should be added as a protected division returning 1 as standard.



Function Set	
Kind of Primitive	*Example(s)*
Arithmetic	`+, *, /`
Mathematical	`sin, cos, exp`
Boolean	`AND, OR, NOT`
Conditional	`IF-THEN-ELSE`
Looping	`FOR, REPEAT`
⋮	⋮

Terminal Set	
Kind of Primitive	*Example(s)*
Variables	`x, y`
Constant values	`3, 0.45`
0-arity functions	`rand, go_left`

Figure 2. Example of primitives in the function and terminal sets [1].

Initialization

In GP a population is initialized by generating a population of trees with the primitives randomly placed as nodes. An initialized population of 3 may appear as in figure 3.

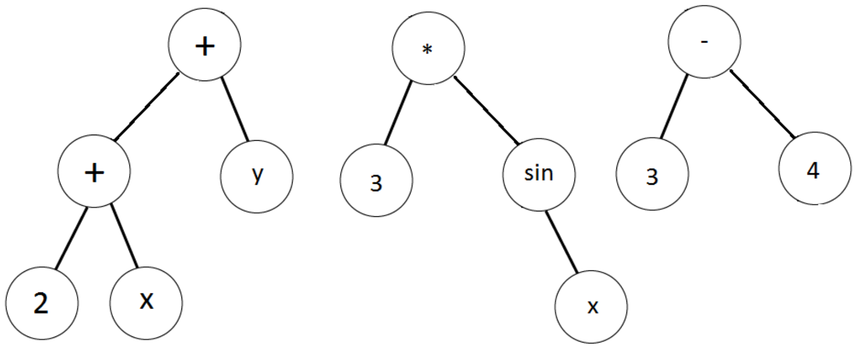


Figure 3. Initialized population with +,,sin,- belonging to the function set and 2,x,3,4 to the terminal set.*

Different initialization methods exist. The *full method* is to build trees where all leaves are at the same depth, and the *grow method* can be used to generate trees of varying depth. Initialization that uses a combination of the two is called *ramped half-and-half*. General for these methods is, that they build trees up to a size of a predefined max depth.

Selection

In GP selection is performed as in other EAs by following a selection method. An often used method is *tournament selection* where individuals are randomly chosen to compete and a superior individual is added to the next generation of individuals. In tournament selection, individuals with a relative high fitness have a good chance, but are not guaranteed, to be selected. Conversely, this means that individuals with relatively low fitness may be selected but with a lower probability. This is known as *automatic fitness scaling* and is a desirable feature of GP. The reason is, that we want to keep some diversity to the population and not let a few early good programs rule out all other inferior programs - these may carry important traits for reaching good solutions.

Fitness
The fitness of a program is a value that says how well the program solves the problem we have defined. As an example, we might be trying to teach a robot to clean our floor, and the fitness could then be how much dirt the robot collects in a given time span. In this kind of problem we would then have to test the program in different test cases. That could be placing the robot in different starting positions or adding obstacles that it should be able to handle. For a GP run the fitness evaluation is often the most time consuming step, as in the example, we would have to wait for the robot to complete its task of collecting dirt in each test case.

Mutation
Mutation happens with a probability that is commonly below 1%[11], and the operation causes one or more nodes of a subtree. For a mutation operation that alters the primitive in a node, it is important to assert that the replacing primitive can actually be inserted at the point of mutation to not corrupt the tree.

Crossover
Analogous to replication in nature, new individuals can be created by crossing the genes of its parents. In GP the crossover operation is carried out by exchanging sub-trees. A traditional crossover technique is *one-point crossover* that selects a single node in one parent and swaps it with a one node in the other. Various other techniques for cross over can be found in the litterature. The probability of a crossover is often around 90%[12].

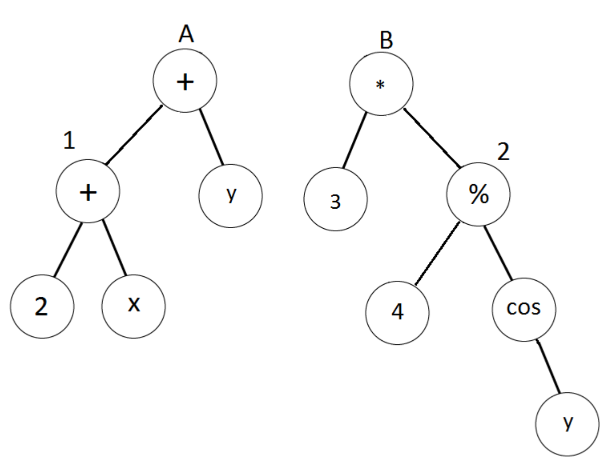


Figure 4. Before one-point crossover at node 1 and 2

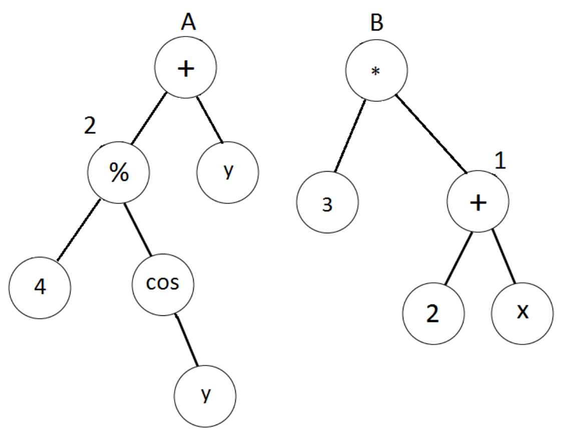


Figure 5. After crossover from figure 4

[11] [1] page 17
[12] [1] page 17

Closure
An important part of performing a GP run is that of closure – i.e. type consistency. The reason for this is that crossover and mutation can make any node end up at any position. To deal with this we can require type consistency in the primitive set - that is, one consistent type of variables, constants, return types and argument types.

Bloat
Bloat is the uncontrolled growth of trees and is something we want to put under control. In an extreme, we can imagine how the crossover and mutation operations may build bigger and bigger trees and that it will make the algorithm slow, build overly complex trees or that we would simply run out of memory.

GP run
A GP run is an execution of Algoritm 1 subject to a set of run parameters. These could be:

- GP techniques[13]
- Population size
- Max tree depth
- Probability of genetic operations
- Max generations
- Fitness goal
- Other control parameters.

GP is a suitable technique for our system and can, as recommended in *A Field Guide to Genetic Programming*, be a good learning experience to implement, which fits well with the motivation for this project. A fair amount of techniques for initialization, crossover and mutation are referenced in the field guide, so our system should allow for easy extension of these. It is recommended that an important part of creating the GP is to be rigid about whether its implementation is working as we think it does. Its random nature and the complex crossing and altering of the trees could easily be subject to misinterpretation and logical errors.

1.3.2 Existing software

At MSDN Magazine, Microsoft exhibits a program called “GeneticAlgorithms”[14] from 2004 written by independent consultant Brian Connolly[15]. The program uses genetic programming to learn an ant to find food distributed in a two dimensional grid. In a nutshell, the program uses reflection to make certain members of an Ant class the primitives of the GP. It is then designed to use .NET’s CodeDom[16] to dynamically create and compile a subclass of the Ant class, representing a given individual of the GPs population. To evaluate the fitness of an ant, an dynamically crated instance of this subclass is and then called a given number of times to make the ant behave, and the fitness is then read from the amount of food it has eaten.

[13] “GP techniques” refer to: initialization, mutation, selection and crossover
[14] http://msdn.microsoft.com/en-us/magazine/cc163934.aspx
[15] http://www.ideajungle.com/
[16] http://msdn.microsoft.com/en-us/library/y2k85ax6.aspx

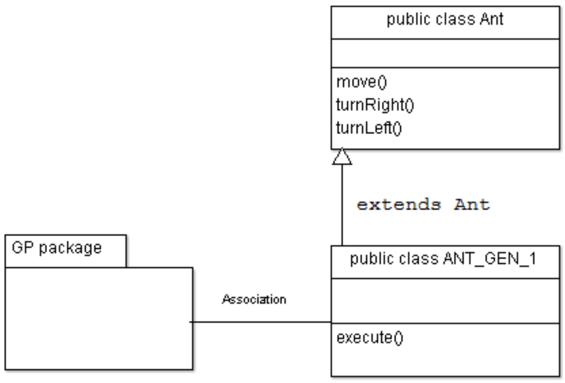
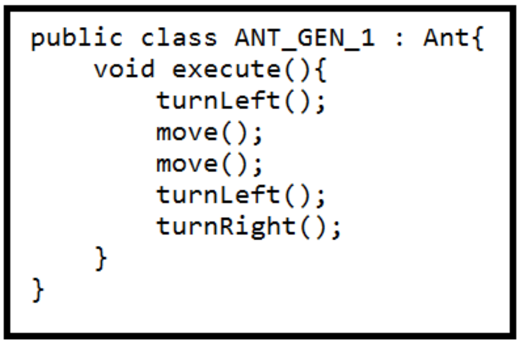


Figure 6. Class diagram of the GP when having created the dynamic ANT_GEN_1 class



```
public class ANT_GEN_1 : Ant{
    void execute(){
        turnLeft();
        move();
        move();
        turnLeft();
        turnRight();
    }
}
```

Figure 7. Example of how the individual's source code may look like in ANT_GEN_1 in figure 6.

The program has a few good ideas we can use as inspiration. Firstly it gets the primitives from a c# class. We can have a conventionalized manner of defining problems in c# that our systems use and then, as intended, use GP as a general problem solver. This way we can define problems by defining them in c#. Another interesting feature is that of being able to see the syntax tree represented in c#, achieved with the CodeDom. This can be a helpful feature for obtaining insight to how the structure of (small) trees evolves and what code is generated Inspired by this we wish to create a system which can read a problem in a simple manner (written as a c# class) and deliver a program which is subject to how the problem defines its fitness measure. So ultimately we can have Algorithm 2.

```
Input: p, A problem that defines available primitives and fitness function
       Param, GP run parameters
Output: A solution

    1.  Initialize GP with param and p
    2.  Return GP.Algorithm 1
```

Algorithm 2

1.3.3 Abbott's Flatland

We can directly translate the nature of Flatland from Abbott's book [2]. Many of the circumstances discussed in the book would be very complex to try to and reason about, so a few of special interest which are possible to imagine in the system are:

- Women are straight lines, and their extremities are like "needles" that may kill others. It is difficult to avoid collision, and this may cause death. All collision can cause this, but colliding with women is particularly dangerous.
- Constant attraction to the south that helps the Flatlanders navigate around.
- The biggest flatlanders are around 11 inches.

- Correspondence between sides and social rank. *Many-sided* are regarded as circles and are of the *Circle* or *Priestly* order, and are of the highest classes in Flatland and considered wise.
- A male child has exactly one more side than his father, if this is not an isosceles triangle. Rules about inheritance of sides are described with several exceptions (chap 3)

To work with these we can do the following:

- When creatures collide some will die, and those that hit the pointy ends of women will have a high chance of dying.
- Gravity in FPE as down will be south in the 2D world.
- A size of 11 inches is probably a bit too big to work well in the simulation, but we can have a height feature on the agents and let them grow to varying sizes.
- We need to be able to generate polygonal shapes of varying sizes when new Individuals are born. A male child will then be generated as a shape with one more side.
- Simplify this at first and let the first part of the rule be equal for all. Assuming that the Flatlanders also have to eat, we can start by trying to have our GP system develop a food-finder, and then move to the behaviors that we get from the book.

1.3.4 Agents

An agent program is a program that acts autonomously and is defined by its behavior. For our discussion we will think of an agent as one of our geometrical shapes in Flatland, but in reality the ideas can be applied to practically any domain and is in [*Salamon – Agent-based models*][4] referred to as agent-oriented programming - as opposed to object-oriented programming. There is no single way of how agents can be modeled, but in an abstract form we can define an agent as a software entity that *perceives* its surrounding environment with the use of sensors, and acts upon its perceptions through *actuators*. Agents will, at their best, behave intelligently in order to achieve some designed objective. This "behave" is what makes the agent "intelligent" and is by Russell and Norvig defined as the *agent function*[Russell, Norvig – Artificial Intelligence – a modern approach] [3]. The agent function is what maps the agent's perceptions into actions, and from this we can derive how well our agent behaves, that is, evaluate its performance. Connecting this with GP performance evaluation will be the fitness of an individual. To reuse our example of a dirt-collecting robot from 1.3.1, the sensor of our agent robot is what makes it capable of perceiving where the dirt is, which could be, sight sensors. Its actuator would be its ability to, say, drive forward or turn, and its agent function will be the program that our GP has produced for us. The environment it will perceive from is the simulation.

In [4] this simple agent is referred to as a *reactive agent.* A reactive agent does not reason about its environment by creating a knowledge base of its world for future decision making, it simply reacts to its perceptions. And though our agent's actions might be big complex chunks of code, this kind of reasoning about our agent's behavior seems good. Salamon [4] describes that even though these types of agents are relatively simple, they may develop complex and intelligent behaviors. He also presents the *subsumption architecture* by Rodney A. Brooks (MIT) as one of the most influential architectures used in reactive agents. The idea is to have several parallel *layers* of behavior, where each layer can trigger a certain reaction from the agent. The layers are ordered by priority with the lowest layers having the highest priority"[17]. We can image our dirt-collecting robot might only have a "look for dirt behavior", but a more advanced version could have a "if low battery, find outlet" at a lower level. The first thing that comes to mind when trying to

[17] In [4] page 31

reason about this architecture is the complexity. It states that behaviors may be interested in conflicting sensor inputs and may try to achieve conflicting goals, and when we want to add new behaviors they will have to be inserted in the hierarchy which will change the agent's overall behavior completely. However, in this complexity the agent may just as well evolve the behavior needed to get by in its environment. As the architecture is simple[18] it is something we can use for our implementation of the agent. See figure 8 for example of the architecture for the advanced dirt-collecting robot.
With this definition of an agent we should be able to make an implementation of an agent program, whose agent function can be generated by our GP implementation.

Layer	Bahavior
3	**if** true **then** move
2	**If** stands on dirt **then** suck
1	**If** low battery **then** find outlet

Figure 8. Subsumption architecture for dirt-collecting robot

1.3.5 Simulator

FPE and XNA can help us with some of the heavy lifting for creating a life like environment. XNA will allow us to build the simulation in a game like manner, and should make it easier to load content, apply graphics and react to keyboard and mouse input. Having virtually no experience with writing games, respective forums of the two can be used as a learning resource.[19] FPE offers a framework that can give us realistic physics and collision detection, and quite importantly, it also has methods for creating polygon shapes based on a given set of vertices. These can also be added to the engine as *bodies,* I.e. objects that react to physics, and these can produce *fixtures* that can collide. With the FPE comes a *Debug View* project that can be used to get a run-time help view on top of a running simulation to gain insight on what is going in the physics engine. Also, the FPE comes with a suite of sample projects that give good examples of how the engine is used.

[18] Salamon also describes a dynamic version of the architecture that deals with the problem that will arise if the agent has many layers. We assume just a few behaviors will be fine for the Flatlanders.
[19] XNA game studio forum at http://forums.create.msdn.com/forums/default.aspx?GroupID=6 *and* FPE http://farseerphysics.codeplex.com/.

1.3.6 Overall Architecture

Based on the above the overall idea for the system is explained in the following. See overall architecture in figure 9.

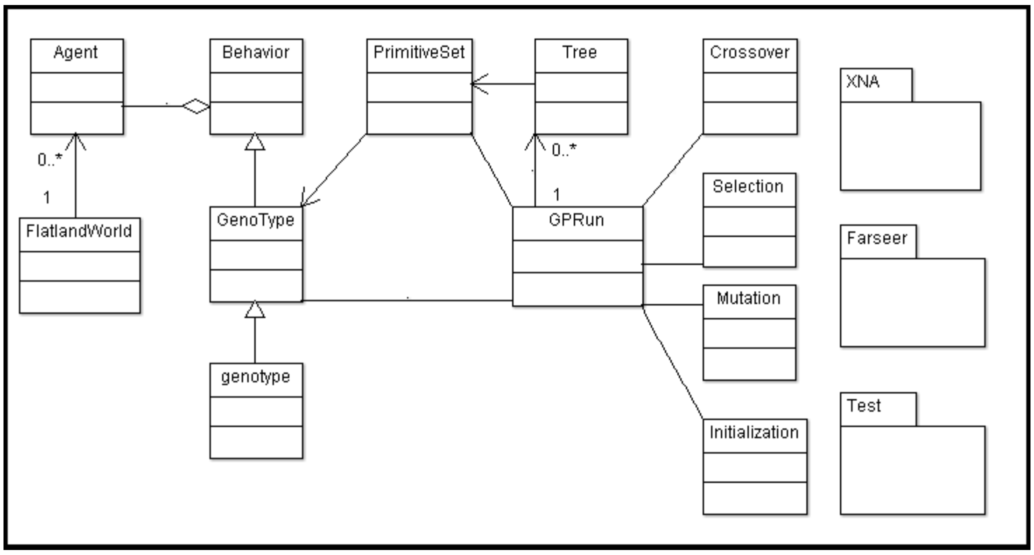


Figure 9. Overall architecture of the system

The FlatlandWorld is the simulation controlling the agents and the environment. An Agent is controlled by its behavior and will just be making its sensors and actuators available. As we see, the agent will aggregate itself *to* a behavior because this is what controls it, and we will be able to read sensors and send commands directly to the agent from the genotype. This is like in figure 8 where the behavior asks for sensor input, and then makes the agent act. In our case, the same thing will happen every time the FlatlandWorld updates the environment; the behavior of the agent will check the agent's sensors and make it act.

The GenoType[20] is the general problem class for the GP class. That is, it is the type we will subclass when defining a problem, which in figure 9 is what the *genotype* class represents. The genotype (with lower case) could be any behavior we want to evolve for the Agent - which we get access to through the hierarchy.

The GP will be initialized by creating a population of trees based on the primitives found in the genotype – idea from 1.3.2 When the run meets its goal test, a solution program will be returned. The GP technique can be replaced with new GP technique, as we found in 1.3.1.

[20] The word genotype is used for our problem defining class, as it is often called this in GP.

2 User guide

This user guide explains the running application as it appears on the included cd. Running the application will present a screen much like in figure 10.

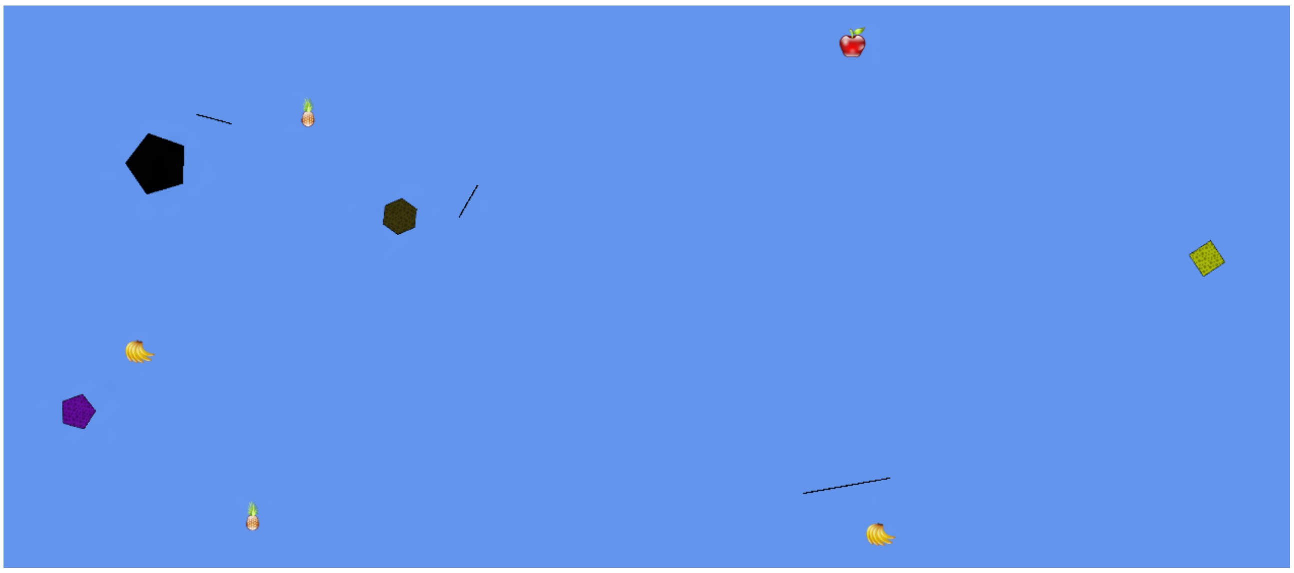
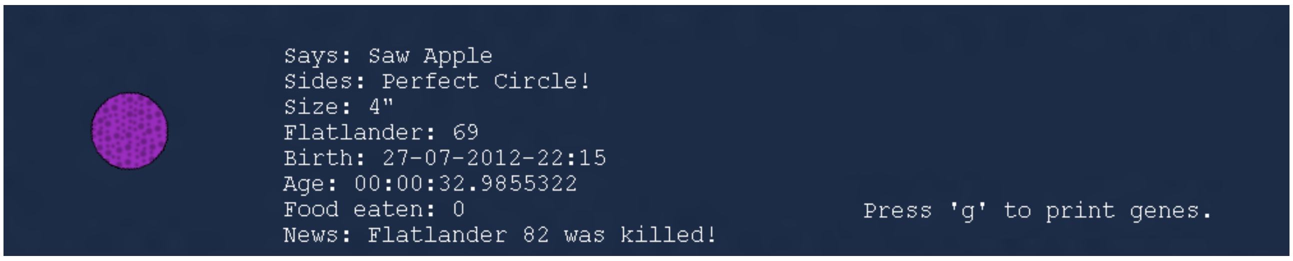


Figure 10 Running the Flatland application

A new population of flatlanders is created every time the application is launched. The geometrical shapes are the male Flatlanders and the lines are the female. The small pieces of food lying around will be eaten if an individual moves into it. A Flatlander is likely to die if he or she hits the pointy ends of a woman, and the same thing could happen when colliding with a male Flatlander, as we described in 1.3.3. The simulation uses gravity to create a small attraction to the south, but their moving around will normally do that this cannot be seen visually. Life and death resolves to not hitting too many other individuals, but still moving enough to find food. Every 10 seconds two randomly chosen individuals will breed a new creature and by crossing and mutating the genes of the parents (with some probability). New individuals will appear at a random place in the world. When all available food has been eaten, each individual will lose a piece of food from their personal supply, and those who run out of food will die.

A mouse click on an individual will toggle camera tracking, and an info panel will appear in the bottom of the screen. See figure 11.



```
Says: Saw Apple
Sides: Perfect Circle!
Size: 4"
Flatlander: 69
Birth: 27-07-2012-22:15
Age: 00:00:32.9855322
Food eaten: 0                          Press 'g' to print genes.
News: Flatlander 82 was killed!
```

Figure 11 Info panel on a Flatlander

The ‘says’ field is the individuals voice, and they will speak notoriously about their experiences as they go through life. A Flatlander’s size can go up to 11 inches (1.3.3) and have up to 25 sides at which point it becomes a circle. This is just an arbitrarily picked number so we won’t be adding shapes to the physics engine that are more complex needed. New born males will always have one more side more than their father (1.3.3), so as the applications runs circle shapes will be in abundance. With the panel open, pressing the ‘g’ key will print the phenotypes along with a mug shot of the selected individual to a folder relative to the application - the path appears on the info panel. Besides the mug shot, a C# and a .Gene file is created. The code in the C# file corresponds to a phenotype object in the .Gene file. This will be explained in 3.4.

Pressing the ‘h’ key will display a general info panel with info on population size, food reserve and other possible inputs. See table 1.

Key	**Description**
h	Help. Show general info panel
z/x	Zoom in/out
Arrow keys	Navigate
p	Print selected Flatlander
f	Throw random amount of food
b	Breed new Flatlander
c	Change world color
q	Quit

Table 1. Keyboard input

3 Technical Description

In this section we discuss the technical implementation of Flatland. We will start by describing the GP class implementation and then the agent. We will see that the GP works in isolation, but that the programs it produces can be used as the behavior of an agent – this is what the architecture in figure 9 seeks to accomplish.

3.1 Genotype

As in figure 9 we create a genotype by sub-classing the GenoType class. The solution offers several examples of how genotypes can be written, and includes a template[21] that declares what is needed as a minimum.
Primitives are defined by using an attribute on the class members that we want to include. In code example 1 we are declaring four terminals and one function, which is the parameterized method. Fields are used as terminal variables and we would often be setting these just before executing the generated code. The generated code will use the field 'b' and the array indexes of 'bools'. Whether the attributed member is a terminal or a function is implicit, and we should just think about what mix of methods and fields we want for the agent. Code example 1 shows what class members it is possible to use with the primitive attribute.

```
[Primitive]
public bool b;

[Primitive]
public bool[] bools = new bool[3];

[Primitive]
public void MoveLeft() {
    Agent.SetDirection((float)Math.PI);
}

[Primitive]
public bool SeesFood() {
    return Agent.SeesFood;
}

[Primitive]
public bool SeesFood(bool b) {
    return Agent.SeesFood && b;
}
```

Code example 1: Declaring primitives in the genotype

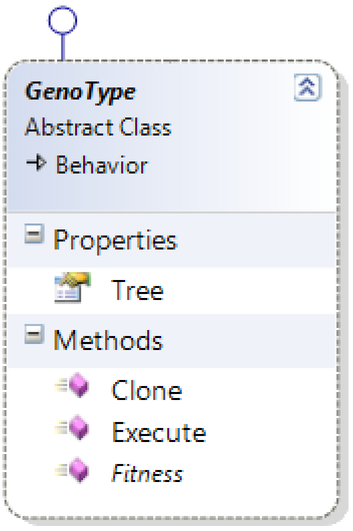


Figure 12 – GenoType class

[21] See appendix B2

In figure 12 the GenoType class is shown. It requires that deriving classes will provide a fitness method, which the GP will be calling during a run. *Execute* and *Tree* are explained in 3.2.

3.1.1 The primitive sets

Getting the primitives from the genotype is handled by the PrimitiveSet class (see figure 9), which will hold the sets during a run. The PrimitiveSet class is also responsible for handling closure (see 1.3.1). Furthermore it is possible to query the PrimitiveSet for a subset, say, only the booleans in code example 1. As we will see in the following, Flatland can handle a primitive set that violates closure.

3.1.2 Command or expression

When writing a genotype to the agent like the one in code example 1, we are mixing Boolean and void terminals. This makes good sense if we think about what kind of control logic we will be adding to the agent, but internally, it will of course add complexity and will have to handle this with care.

When we create the kind of command logic we did in code example 1 the PrimitiveSet class will add a conditional *if-then-else* to the function set together with a *sequence* function – representing a sequence of functions or terminals. These are of course needed to make it possible to create some interesting code for the agent. Analogous to the reasoning in the intro of 3.1, it is a design choice that these two functions are removed from the implementer of the agent logic – it is attended to be clean and take focus away from thinking about 'GP logic' and think of agent problems instead.

As an alternative to the above, genotypes have an option of implementing an interface *IExpression<T>*. When doing so, the closure type will be that of T, and it is possible to use any combination of methods adhering to type T. This would for example be used for arithmetic or logical problems and leaves us with a great deal of flexibility. In using *IExpression<T>* however, it should be no surprise, that we can't use void, and the *if-then-else* and *sequence* functions won't be added to the function set.

When we need to distinguish, these will be referred to as Command genotype and IExpression<T> respectively.

3.2 Tree

The tree data structure (figure 9) is based on a composite pattern and its class diagram is depicted in figure 13.

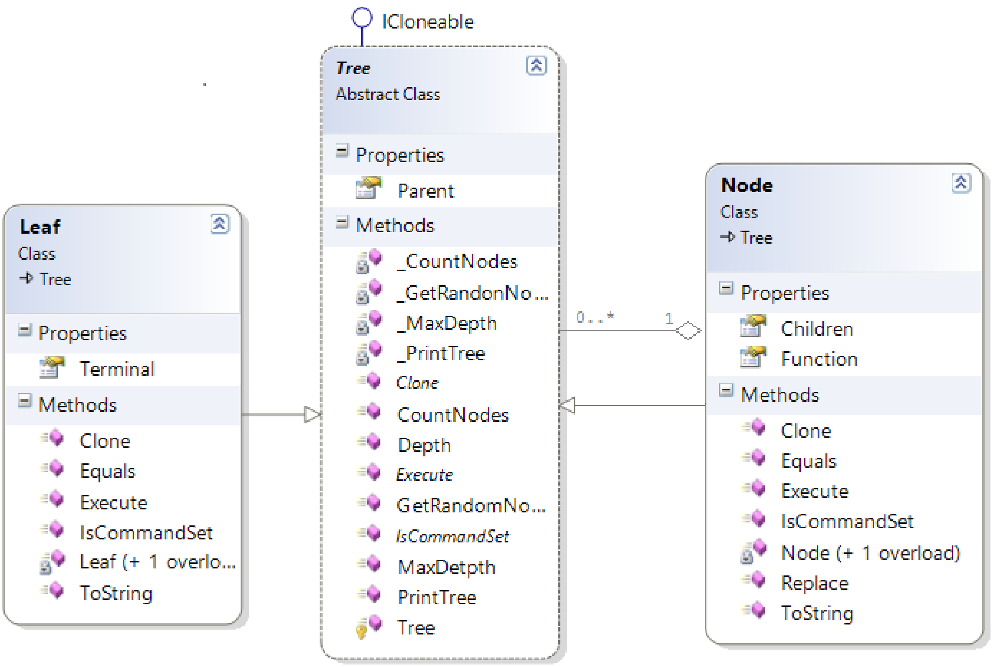


Figure 13 Class diagram of the Tree

Every leaf carries a reference to a Terminal and every node a reference to a *Function*. To interpret[22] a tree we will call *Execute* on its root node with a reference to the genotype (being the object we got the primitives from). The tree will then recursively interpret itself by looking up the primitives on the genotype, and then invoke these with reflection. For the kind of command logic problems we saw before the interpretation will end at some arbitrary leaf, and for the IExpression<T> a value T will be returned at the root.

Equals on tree has a significant meaning: if two nodes are equal it means that they are interchangeable. This is used by the crossover and mutation to operate on the tree, to know if they can replace a given node. The node will just pass on the equals to the relevant Function or Terminal that will answer if a replacement is possible. While this is not bullet proof, it allows us to use the mixed voids and Booleans. The relevant unit test can be found in the TreeTest class.

3.3 GP

The GP class encapsulates the GA. It is constructed with a genotype and optionally the run parameters we listed in 1.3.1 (GP run). The GP will be calling the fitness method on our genotype that it inherited from the GenoType (figure 12). When the GP run terminates, we are returned the program that is the best solution it has found. We use the returned solution by calling the execute method explained in 3.2.

```
GP gp = new GP(new MyGenoType());
MyGenoType solution = (MyGenoType)gp.Run();
solution.Execute();
```

Code example 2: Using the GP

When the GP run terminates it will print a run result to excel, or a text file if excel is not installed. The run result shows a graph of how the max fitness value and average fitness value in the population have been evolving during the run[23].

3.4 GP Techniques

Flatland offers different GP techniques that can be chosen when using the GP class. In the Flatland implementation the intention is to make it easy to reuse the system with a new GP technique. In addition, we also want to make it difficult for future GP techniques to break anything. To achieve this a few things are done. From the GP class, all GP techniques are called through a proxy, as seen with initialization in figure 14.

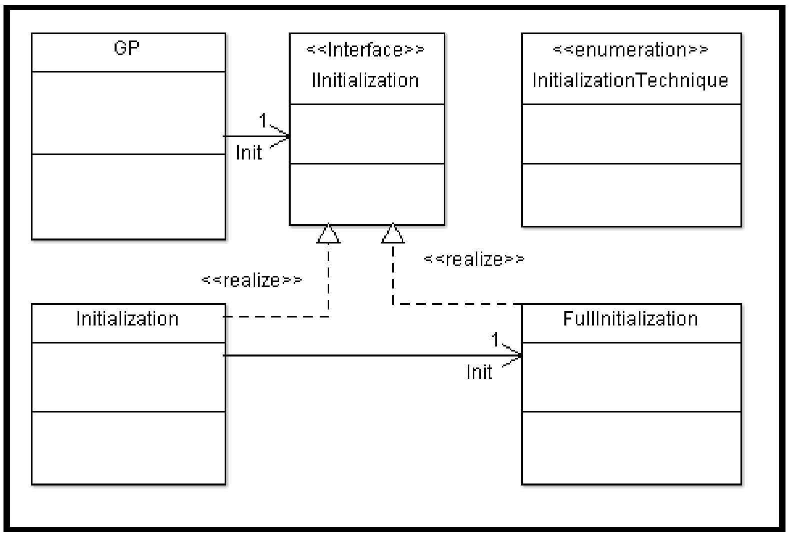


Figure 14 Initialization

[23] See Appendix C1 for examples.

With this we will not have to modify anything in the GP class when adding new GP techniques. Then, if we want to add a new GP technique, for example an initialization technique, we do it in a bit of a novel way. We implement IInitilization, and then we let the system know about us by setting our type as an attribute on an InitializationTechnique[24]. By doing so, the system will find us with reflection, and our new initialization technique will automatically be included in relevant unit tests. This is done by enumerating the InitializationTechnique enum. The idea is that what the unit test requires is also what new components will be tested against when entering the system. This design is consistent for all GP techniques.

The following GP techniques are currently available in Flatland:

Initialization (1.3)

1. Full
2. Grow
3. Ramped half and half

Full initialization is capable of creating trees that deals with the closure problem in 3.1.2. Both Grow and Ramped half and half reuse the implementation in Full – it takes an optional grow argument. Its uses an additional build method for dealing with the Command genotype. This method can handle the voids, *if-then-else,* and *sequence,* and it will use the same build method as IExpression<T> for Boolean sub-trees, by constructing it with a Boolean subset from the PrimitiveSet (3.1.1).

Mutation

1. Node replacement
2. Sub-tree mutation

The mutation methods are as we would expect. Node replacement will replace a random function with a random function, or random terminal for random terminal. Sub-tree mutation uses Grow initialization to generate a new sub-tree.

Selection (1.3)

1. Tournament

Tournament takes two random individuals from the population and lets the fittest live on in the next generation. It will always generate a population of the same size as the previous, and individuals may be selected multiple times.

Crossover

1. One point crossover

One point crossover combines random picking of nodes with methodic traversal of one tree to find a crossover point. It uses equals on the nodes to assure that a crossover can be performed.

Bloat

A final, but important, feature is bloat control (3.1) which is done with *size fair crossover* and *size fair mutation*[25]*.* This works by checking that a given operation on a tree won't make it too big. Crossover and Mutation are passed the max tree depth from their proxies and they must be able to control bloat to pass their unit tests.

[24] See appendix B3 for the InitializationMethod

[25] [1] page 105

3.4 Evolving behaviors

With the GP implementation we can begin training our agents. And to do so we will have to be able to return a fitness value when called by the GP. This, however, poses a problem. Evaluating the fitness of the agent is not trivial, as we want to evaluate it in the game loop. It would require some messy asynchronous handling to receive the fitness call in the genotype and then go evaluate the agent for a given time, before returning the fitness. To accommodate for this, the GP can be used in a *step-wise* and *not fitness-evaluating* manner. By allowing this we can pick and evaluate each individual in turn and evaluate it with an agent, and then ask the GP to step when we are through the population. The logic for putting this crank on the GP is encapsulated in the *FlatlandLab* class. FlatlandLab can be handed to Flatland which will setup at closed training-environment where the agent can be trained.

```
GenoType genoType = new MyGenoType();
FlatlandLab lab = new FlatlandLab(new GP(genoType));
FlatlandWorld = new FlatlandWorld(lab);
FlatlandWorld.Run();
```

Code example 3: Training the behavior of an agent from a genotype

When the evaluation terminates the simulator closes, and the GP will leave us a .Gene file just like the one we got in 2. The .Gene file is the evolved behavior of the agent, i.e. a tree at a given time. In 2 we handpicked this from a specimen in the Flatland simulation, in the above we have had the GP produce us one subject to a fitness goal. As the above example would only create an environment with the physics of Flatland and an agent, we would normally sub-class the FlatlandLab class and use its virtual *ResetEnvironment* method to set our training data in the environment. The solution includes a *FoodLab* that demonstrates this, and it can be used to train a food finding agent. The idea with this is that we can handpick, train and spawn entire populations from the genotypes, which gives a game-like and lively approach to what Flatland can be used to.

When starting Flatland, we can use these evolved genotypes (phenotypes) as the *initial* genotype for all the agents in Flatland, and see them evolve it further. We do this by inserting either a .Gene file or any other genotypes in the layers of the agent as shown in code example 4. For the system to be able to find the .Gene file, it has to be placed in the GeneBank folder.

```
public enum Layer {

    [Subsume(typeof(Grow))]
    GROW,

    [Subsume(typeof(Skin))]
    SKIN,

    [Subsume("FindFood")]
    FIND_FOOD,
}
```

Code example 4: Adding genotypes to the agent's layers

The raw enum value corresponds to the layer value from figure 4. When the Flatland game instance is loaded, a behavior factory will get the GenoType for each layer, and put them together following a chain-of-responsibility pattern, which is implemented by the Behavior class (see figure 9). This means, that when we are implementing a new GenoType for the agent, we have the possibility of breaking the chain. E.g. if we are *low on battery*, then we break the chain and execute the *find outlet* program. Here the *find outlet* program, is a program we have evolved from a GenoType. See figure 17 for flow chart.

3.5 Agents

When an agent, or Flatlander, is born, it is given a number of vertices relative to its parents. These vertices are then used to calculate its actual shape. We do this by selecting a random point on the unity circle and then move a distance of 2 * pi / vertices on the perimeter for each vertex the agent should have.

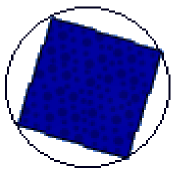


Figure 15: Agent being shaped

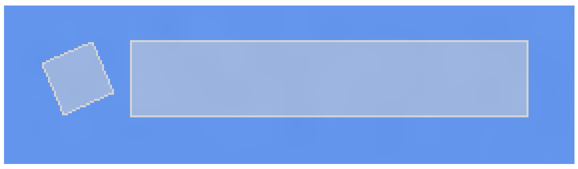


Figure 16: Agent in debug view

After the agents have been shaped, we use the generated vertices to add the agent to the physics engine, so that we can receive events when the agent collides with other objects. All agents are added with a direction vector that initially point in the same direction irrespective of how they were shaped. This will make them appear as if they can have their front at different places and give a bit of nuance to the agent.

The primary sensor of an agent is its sight and body sensor which can be seen on figure 16. Figure 16 is a screen shot taken from using the debug view in FPE, which allows us to see all fixtures[26] that exist in the physics engine – for obvious reasons sight has to be invisible to us. The long column is what the agent can actually see, and it will always point in the direction that the agent is moving.

Its sensors tell if the agent sees food or other agents, and its actuators are steering the agent by adjusting its direction or speed. A flow chart for how these work with the behavior can be seen in figure 17.

[26] A Fixture is an object that FPE uses. By creating a fixture with the FPE we can use event handlers to react on collision.

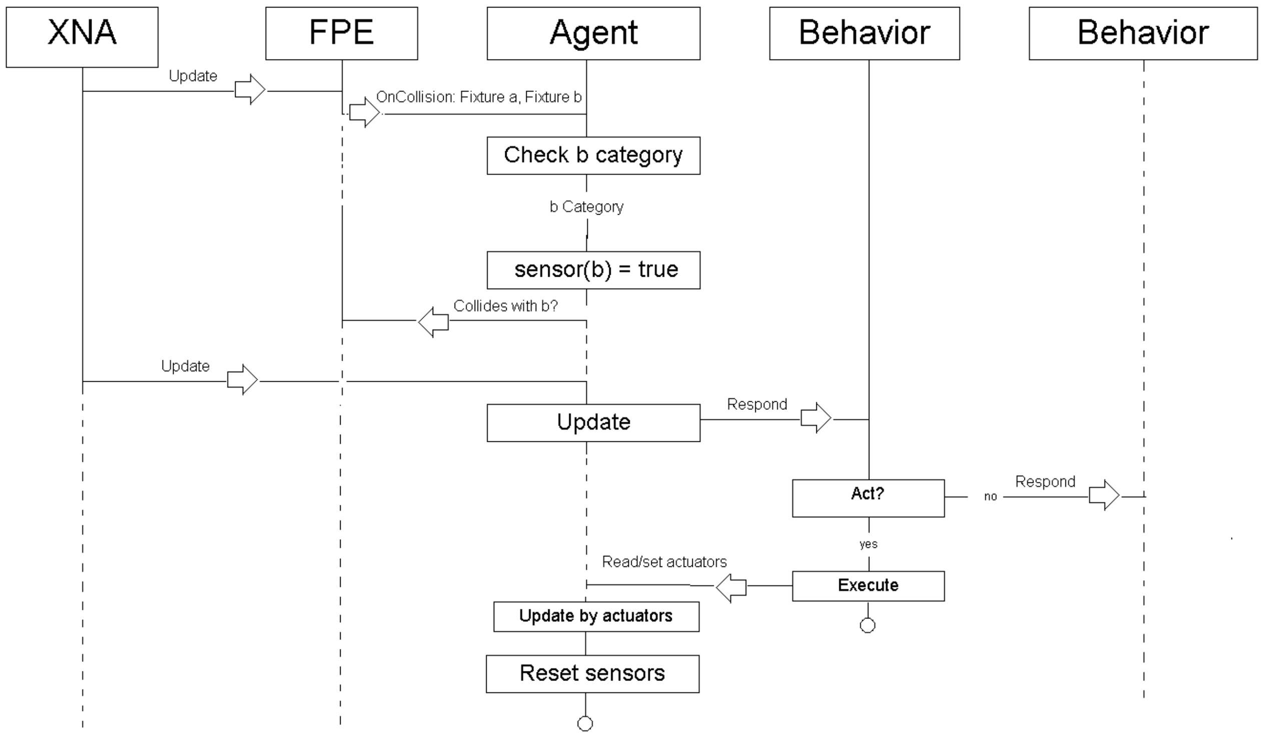


Figure 17. Agent controlled by its behavior.

Though female agents appear as simple lines, they are actually very thin rectangles. Collision detection seems much better when using a thin rectangle and the visual outcome is nearly identical. Women have an extra sensor at each extreme, which upon collision may kill another agent.

General for the sensors is that they are made by adding them as independent objects in FPE. The good thing about doing it this way is that it we get efficient collision detection almost for free, and besides having to calculate the position of the sensors in relation to the agent, we can use them right away. On the downside we are faced with the problem that the agent sensors now collide with the agent itself and that we are adding a lot of bodies to the physics engine, which eventually will make it slow. The solution to this has been to accept the latter, and place the sensors just off the body of the agent to avoid self collision.

The color of an agent is decided by a *Skin* genotype that can yield any color. One way to see this is to harvest the genes of an agent (explained in 2) and then use the Skin.Gene file on a layer. This would cause the whole population to have the same color at first, but changing as they breed. The texture of their skin is manipulated by the SkinTexture genotype in combination with a texture found in the MyTextures folder. Any textures added to this folder might be used as a base texture that the SkinTexture might manipulate. Besides adding colors and textures to the Flatlanders, these genotypes are mainly meant to be an example of the many ways one can experiment with the system.

The agents ability to find food is evolved using FlatlandLab and a very simple genotype with basic features like move, stop and turn[27]. Evolving the behavior is quite easy, and with the right run parameters it is often possible to create a fair food-finder in the first generation. The randomness of the GA will be able to hit a simple combination of methods that result in a useful pattern for the agent to find food.

[27] See Appendix C2 for FindFood genotype and C3 for generated program

4 Test

In this section the test of the system is described. In [1] on testing a GP it states that it requires "*the will to thoroughly test the resulting system until it behaves as expected. This is actually an extremely tricky issue in highly stochastic systems such as GP …*"[28] - which has also been the case with this implementation. The unit test for Flatland can be found in the "Test" project. The test cases conducted here resolve to observations and a documented run that should be reproducible. 3.5 and 3.6 this section will comment on the state of the system

4.1 Test run

Description:
A GP run given bad conditions to solve a simple problem. We do this to make it easy for ourselves to understand the problem and what solution to expect, but converresely we want the GP show that it can optimize for a perfect solution. We set a small population size, and set it a relative big max depth compared to the simple nature of the expression needed to solve the problem. The template for this problem is in ../Flatland.GP/Examples/MathExample.cs. It uses IExpression<int>.

Input:
Function set: +, -, *, %, /
Terminal set: X
Population: 5
Max tree depth: 15
Max generations: 1000
Method: Ramped half and half, one point crossover, sub-tree mutation, tournament
Parameters: X = 42
Fitness goal = X*X*X
Fitness = Fitness goal – (abs (fitness goal - X*X*X))

Expected output:
Increasing max fitness.
Maybe an increasing avg. fitness, or at least some tendency.
An overly complex program with exact fitness.

[28] [1] page 147

Result:

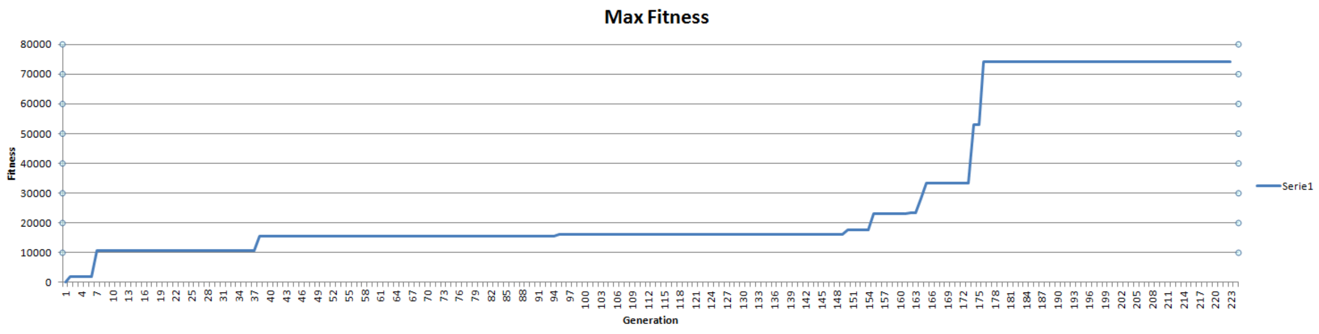


Figure 18. Max fitness graph from GP run of running MathExample.cs

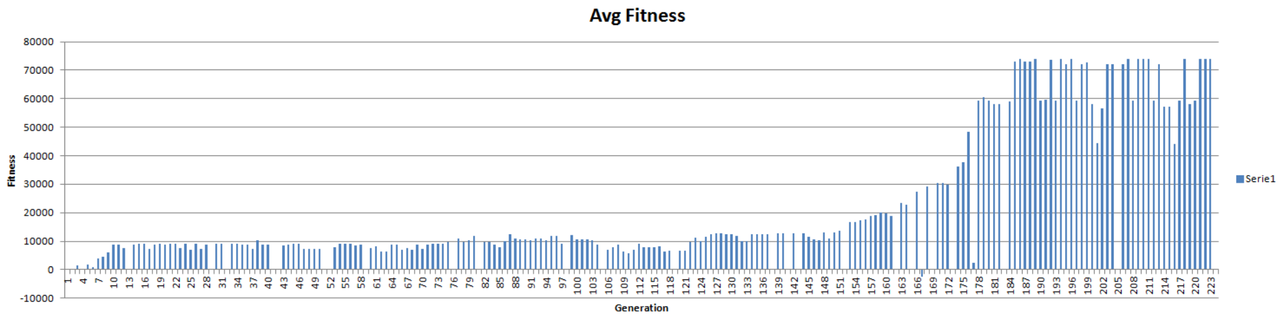


Figure 19. Average fitness graph from GP run of MathExample.cs[29]

Comments:

Max fitness ok. Final pleatue was just 1 from fitness goal. Avg fitness graph output was unreadable due to big negative values, these were set to 0 for the graph in figure. Program source code is in Appendix A6

[29] See Appendix A5 for actual values.

4.2 GP run Test

The GP class is our biggest concern and should be checked thoroughly. Its tests require that for each step during a run:

1. No individual is referenced to more than once.
2. No node is referenced to more than once.
3. Nodes have a parent (except root node) that knows them as a child.
4. Nodes hold a Function and leaves hold a Terminal.
5. No tree is deeper than max depth.
6. Every tree can be interpreted.
7. The above holds for a run with any combination of GP techniques.

In addition, exploratory testing has been done continuously during development. This has been done by inspecting the prints from the test project or the debugger using object watch and object id[30]. The figures in the following are transcriptions of output from a few of the trees produced by the test methods[31]. They are provided here as documentation.

4.3 Initialization

Description:
Documented observations of trees after initialization and their desired shapes. Test method is ThreeDepthTrees in the InitializationTest.cs

Input:
An IExpression<T> problem and a command problem.
Expected output:
Trees with their respective characteristics and corresponding source code.
Result:
The trees appear to be initialized correctly in terms of shape and are in correspondence with the generated code. Experienced compile error because of private field in template. Crash in test methods from last refactoring. See Figure 11, 12 and 13 for examples.

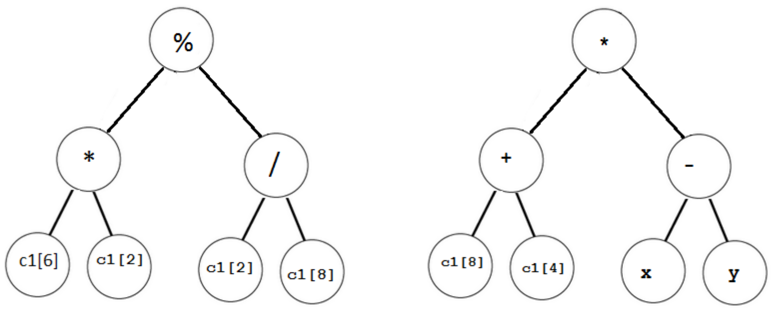


Figure 19. Full initialization examples. c1 is an integer array.
Problem class: TestGene.cs Transcribe of Appendix A1,A2 Full 1 and Full 2.

[30] http://blogs.msdn.com/b/zainnab/archive/2010/03/04/make-objectid-vstipdebug0015.aspx
[31] See Appendix A1 to A6 for output.

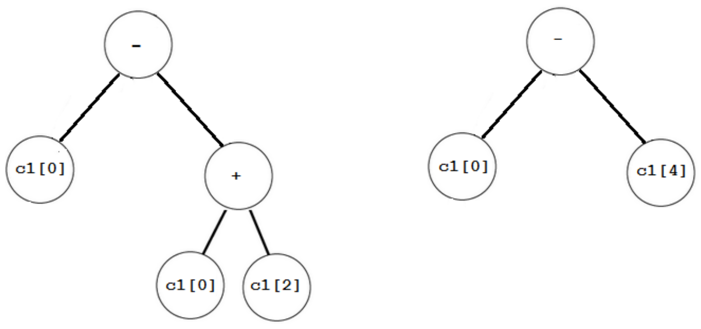


Figure 20 Grow initialization examples. c1 is an integer array
Problem class TestGene.cs Transcribe of Appendix A1, A2 Grow 1 and Grow 2

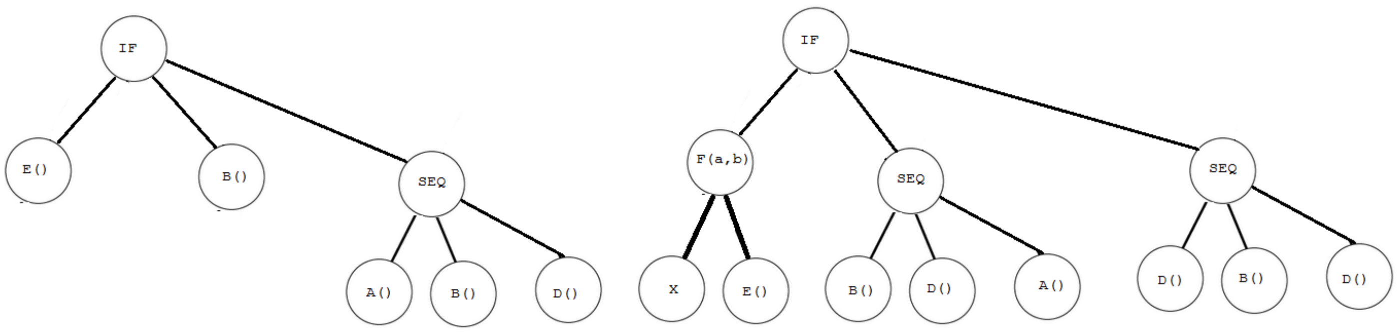


Figure 13 Ramped half and half initialization example
IF is an if conditional and SEQ is a sequence of statements.
E() is a boolean zero arity method, F(a,b) is a boolean method with two boolean parameters, X is a Boolean variable.
The rest are void methods.

4.4 Crossover

Description:
Observations of trees before and after crossover and the desired effect. Test methods are CrossOverExpressionTrees and CrossOverCommandTrees test method in the CrossOverTest.cs

Input:
2 expression problems and 2 command problems with all different named primitives to make it easier to find the nodes that have changed. Crossover rate of 1.

Expected output:
Identifiable crossover of subtrees.

Result:
Ok, see Apendix A3.

4.5 Mutation

Description:
Documented observations of trees before and after mutation and the desired effect. Test method is MutationsTest test method in the MutationsTest.cs

Input:
An expression problem and a command problem. Mutation rate of 1
Expected output: Trees with 1 or 0 visible mutations. Occurrences in both nodes and leaves.

Result:
Ok, included results of root for root, and terminal replacing function see Appendix A4

4.6 Unit test

The existing unit tests pass and give assurance that the system is unlikely to crash during runs. During development it has not been uncommon that an invalid tree would be generated, but to some luck, these almost always cause immediate crashes.

Most unit tests produce printouts of the trees, which can be very helpful. One often wants to inspect the tree and look for certain outcomes, however, this can also be deceiving and should be used with caution.

The idea with automatically adding new GP techniques to the tests has it pros but also its cons. The test cases are growing rather complex and errors may be difficult to understand. It is, however, a must that we cross these tests

4.7 Test result

As we found in 1.3.1 some rigidness is needed to convince ourselves that the system works as we think it does. Some work has been done, but to say anything with certainty seems hard. As shown in 4.1 it can easily crack some simple problems, but the increasing fitness value and fitness average does not prove that we are doing everything the right way. Taking the food finder as an example, which is easy to get a result from, the same thing could be accomplished with just complete randomness, which would be against the idea of trading certain traits to yield good solutions.

5 Conclusion

We have seen a system that seeks to give Flatland the best possible conditions for life. In realizing Flatland we have translated a few characteristics from the novella into rules and characteristics of a virtual world, and made a simple food finder with a machine learning technique.

The combination of the techniques and theories found during the analysis, have been used to try and shape a solution were many ideas have had to intermix. Flatland has its dents and peculiarities from this, but in general it offers reusability and functionality that others can use.

The solution is the creative answer to a problem of endless complexity that could have taken many other directions. In considering the problem statement the implicit question of *"what to do with it?"* has become the vision of something playful and reusable, and with that realize flatland with artificial life. Other directions that may have yielded more impressive results, would have been against the motivation behind the project.

List of figures

Bibliografi

[1] R. Poli, W. B. Langdon og N. F. McPhee, A Field Guide to Genetic Programming, San Francisco: Creative Commons, 2008.

[2] E. A. Abbott, Flatland. A Romance of Many Dimensions, Basil Blackwell. Oxford, 1884.

[3] S. Russell og P. Norvig, Artificialo Intelligence, A modern approach, New Jersey: Pearson, 2009.

[4] T. Salamon, Agent-Based Models, Tomas Bruckner, 2011.

Online resources

http://www.genetic-programming.com/
http://www.grammatical-evolution.org/
http://ncra.ucd.ie/geva/geva.pdf
http://www.obitko.com/tutorials/genetic-algorithms/index.php

http://msdn.microsoft.com/en-us/library/gg145045

http://avida.devosoft.org/

Appendix A1

Test output from `ThreeDepthTrees in InitializationTest.cs.`

ThreeDepthTrees : Passed

```
Init method: Full 1
Tree:  0
Depth: 1 - Tree:  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 11653293
Depth: 2 - Tree:  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 54636159
Depth: 2 - Tree:  Int32 Div(Int32, Int32)Child count: 2 HashCode: 42708074
Depth: 3 - Tree: ARRAY_INDEX: 6 HashCode: 47633461
Depth: 3 - Tree: ARRAY_INDEX: 2 HashCode: 63130991
Depth: 3 - Tree: ARRAY_INDEX: 2 HashCode: 22322349
Depth: 3 - Tree: ARRAY_INDEX: 8 HashCode: 64981649

//From A2 Full 1 this.Mod(this.Mul(c1[6], c1[2]), this.Div(c1[2], c1[8]))


Init method: Full 2
Tree:  1
Depth: 1 - Tree:  Int32 Div(Int32, Int32)Child count: 2 HashCode: 38493088
Depth: 2 - Tree:  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 54467399
Depth: 2 - Tree:  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 35113868
Depth: 3 - Tree: ARRAY_INDEX: 2 HashCode: 36620214
Depth: 3 - Tree: ARRAY_INDEX: 7 HashCode: 37296927
Depth: 3 - Tree: ARRAY_INDEX: 7 HashCode: 640117
Depth: 3 - Tree: ARRAY_INDEX: 4 HashCode: 28805302

// From A2 Full 2 this.Div(this.Mod(c1[2], c1[7]), this.Mul(c1[7], c1[4]));


Init method: Full 3
Tree:  2
Depth: 1 - Tree:  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 21170186
Depth: 2 - Tree:  Int32 Add(Int32, Int32)Child count: 2 HashCode: 13134304
Depth: 2 - Tree:  Int32 Sub(Int32, Int32)Child count: 2 HashCode: 54172779
Depth: 3 - Tree: ARRAY_INDEX: 8 HashCode: 21855962
Depth: 3 - Tree: ARRAY_INDEX: 4 HashCode: 43994237
Depth: 3 - Tree: VARIABLE: X HashCode: 33583636
Depth: 3 - Tree: VARIABLE: Y HashCode: 34868631

// From A2 Full 3 return this.Mul(this.Add(c1[8], c1[4]), this.Sub(X,
Y));


Init method: Full 4
Tree:  3
Depth: 1 - Tree:  Int32 Sub(Int32, Int32)Child count: 2 HashCode: 25584554
Depth: 2 - Tree:  Int32 Add(Int32, Int32)Child count: 2 HashCode: 10454271
Depth: 2 - Tree:  Int32 Div(Int32, Int32)Child count: 2 HashCode: 680171
Depth: 3 - Tree: ARRAY_INDEX: 6 HashCode: 30607723
Depth: 3 - Tree: ARRAY_INDEX: 7 HashCode: 35170261
Depth: 3 - Tree: VARIABLE: Y HashCode: 39157888
Depth: 3 - Tree: ARRAY_INDEX: 1 HashCode: 17274517
```

// From A2 Full 4 return this.Sub(this.Add(c1[6], c1[7]), this.Div(Y, c1[1]));

Init method: Grow 1

```
Tree:  0
Depth: 1 - Tree:  Int32 Sub(Int32, Int32)Child count: 2 HashCode: 39155797
Depth: 2 - Tree: ARRAY_INDEX: 0 HashCode: 17180427
Depth: 2 - Tree:  Int32 Add(Int32, Int32)Child count: 2 HashCode: 34921712
Depth: 3 - Tree: ARRAY_INDEX: 0 HashCode: 27973187
Depth: 3 - Tree: ARRAY_INDEX: 2 HashCode: 50833863
```

// From A2 Grow 1 return this.Sub(c1[0], this.Add(c1[0], c1[2]));

Init method: Grow 2

```
Tree:  1
Depth: 1 - Tree:  Int32 Sub(Int32, Int32)Child count: 2 HashCode: 5822459
Depth: 2 - Tree: ARRAY_INDEX: 0 HashCode: 60684095
Depth: 2 - Tree: ARRAY_INDEX: 4 HashCode: 46429731
```

// From A2 Grow 2 return this.Sub(c1[0], c1[4]);

Init method: Grow 3

```
Tree:  2
Depth: 1 - Tree:  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 8963119
Depth: 2 - Tree: ARRAY_INDEX: 5 HashCode: 687191
Depth: 2 - Tree: ARRAY_INDEX: 2 HashCode: 30923613
```

//From A2 Grow 3 return this.Mul(c1[5], c1[2]);

Init method: Grow 4

```
Tree:  3
Depth: 1 - Tree:  Int32 Add(Int32, Int32)Child count: 2 HashCode: 49385318
Depth: 2 - Tree: VARIABLE: Y HashCode: 7746814
Depth: 2 - Tree: VARIABLE: Y HashCode: 13062350
```

//From A2 Grow 4 return this.Add(Y, Y);

Init method: RampedHalfAndHalf 1

```
Init method: RampedHalfAndHalf
Tree: 0
Depth: 1 - Tree: IF_THEN_ELSE Child count: 3 HashCode: 38600745
Depth: 2 - Tree: Boolean E() HashCode: 59311937
Depth: 2 - Tree: Void B() HashCode: 51791499
Depth: 2 - Tree: SEQUENCE Child count: 3 HashCode: 48916090
Depth: 3 - Tree: Void A() HashCode: 53740413
Depth: 3 - Tree: Void A() HashCode: 2399524
Depth: 3 - Tree: Void D() HashCode: 40869743
```

//From A2 RampedHalfAndHalf 1

```
public virtual void Execute() {
            if (this.E()) {
                this.B();
            }
            else {
                this.A();
                this.A();
                this.D();
            }
        }
```

Init method: RampedHalfAndHalf 2

```
Tree: 1
Depth: 1 - Tree: IF_THEN_ELSE Child count: 3 HashCode: 27199126
Depth: 2 - Tree: Boolean F(Boolean, Boolean)Child count: 2 HashCode: 16001149
Depth: 2 - Tree: SEQUENCE Child count: 3 HashCode: 48963094
Depth: 2 - Tree: SEQUENCE Child count: 3 HashCode: 55855606
Depth: 3 - Tree: VARIABLE: X HashCode: 30474330
Depth: 3 - Tree: Boolean E() HashCode: 29167576
Depth: 3 - Tree: Void B() HashCode: 37472519
Depth: 3 - Tree: Void D() HashCode: 8541776
Depth: 3 - Tree: Void A() HashCode: 48835636
Depth: 3 - Tree: Void D() HashCode: 50119998
Depth: 3 - Tree: Void B() HashCode: 40807399
Depth: 3 - Tree: Void D() HashCode: 24393646
```

//From A2 RampedHalfAndHalf 2

```
        public virtual void Execute() {
            if (this.F(X, this.E())) {
                this.B();
                this.D();
                this.A();
            }
            else {
                this.D();
                this.B();
                this.D();
            }
        }
```

Init method: RampedHalfAndHalf 3

```
Tree: 2
Depth: 1 - Tree: IF_THEN_ELSE Child count: 3 HashCode: 23972246
Depth: 2 - Tree: VARIABLE: Y HashCode: 5009246
Depth: 2 - Tree: IF_THEN_ELSE Child count: 3 HashCode: 24089510
Depth: 2 - Tree: IF_THEN_ELSE Child count: 3 HashCode: 10286142
Depth: 3 - Tree: VARIABLE: X HashCode: 60223214
Depth: 3 - Tree: Void B() HashCode: 25690090
Depth: 3 - Tree: Void B() HashCode: 15203380
Depth: 3 - Tree: VARIABLE: X HashCode: 13063473
Depth: 3 - Tree: Void A() HashCode: 50985389
Depth: 3 - Tree: Void A() HashCode: 12641169
```

//From A2 RampedHalfAndHalf 3

```
        public virtual void Execute() {
            if (Y) {
                if (X) {
```

```
                this.B();
            }
            else {
                this.B();
            }
        }
        else {
            if (X) {
                this.A();
            }
            else {
                this.A();
            }
        }
    }
```

Init method: RampedHalfAndHalf 4

```
Tree: 3
Depth: 1 - Tree: SEQUENCE Child count: 3 HashCode: 31981697
Depth: 2 - Tree: SEQUENCE Child count: 3 HashCode: 29890231
Depth: 2 - Tree: SEQUENCE Child count: 3 HashCode: 2883140
Depth: 2 - Tree: SEQUENCE Child count: 3 HashCode: 62632450
Depth: 3 - Tree: Void A() HashCode: 66996861
Depth: 3 - Tree: Void D() HashCode: 62068736
Depth: 3 - Tree: Void A() HashCode: 41629729
Depth: 3 - Tree: Void D() HashCode: 61398511
Depth: 3 - Tree: Void A() HashCode: 11469592
Depth: 3 - Tree: Void A() HashCode: 46369596
Depth: 3 - Tree: Void B() HashCode: 6257078
Depth: 3 - Tree: Void B() HashCode: 13133063
Depth: 3 - Tree: Void D() HashCode: 54116930
```

```
//From A2 RampedHalfAndHalf 4
        public virtual void Execute() {
            this.A();
            this.D();
            this.A();
            this.D();
            this.A();
            this.A();
            this.B();
            this.B();
            this.D();
        }
```

Appendix A2

Source code output relative to output in Appendix A1.

Full 1

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify;
    
    
    [System.Serializable()]
    public class ThreeTreeTestFull_0 : TestGene {
        
        public ThreeTreeTestFull_0() {
        }
        
        public virtual object Execute() {
            return this.Mod(this.Mul(c1[6], c1[2]), this.Div(c1[2], c1[8]));
        }
        
        public override object Clone() {
            return new ThreeTreeTestFull_0();
        }
    }
}
```

Full 2

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify;
    
    
    [System.Serializable()]
    public class ThreeTreeTestFull_1 : TestGene {
        
        public ThreeTreeTestFull_1() {
        }
        
        public virtual object Execute() {
            return this.Div(this.Mod(c1[2], c1[7]), this.Mul(c1[7], c1[4]));
        }

        public override object Clone() {
            return new ThreeTreeTestFull_1();
        }
    }
}
```

Full 3

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify;


    [System.Serializable()]
    public class ThreeTreeTestFull_2 : TestGene {

        public ThreeTreeTestFull_2() {
        }

        public virtual object Execute() {
            return this.Mul(this.Add(c1[8], c1[4]), this.Sub(X, Y));
        }

        public override object Clone() {
            return new ThreeTreeTestFull_2();
        }
    }
}
```

Full 4

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify;


    [System.Serializable()]
    public class ThreeTreeTestFull_3 : TestGene {

        public ThreeTreeTestFull_3() {
        }

        public virtual object Execute() {
            return this.Sub(this.Add(c1[6], c1[7]), this.Div(Y, c1[1]));
        }
        public override object Clone() {
            return new ThreeTreeTestFull_3();
        }
    }
}
```

Grow 1

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify;


    [System.Serializable()]
    public class ThreeTreeTestGrow_0 : TestGene {

        public ThreeTreeTestGrow_0() {
        }

        public virtual object Execute() {
            return this.Sub(c1[0], this.Add(c1[0], c1[2]));
        }

        public override object Clone() {
            return new ThreeTreeTestGrow_0();
        }
    }
}
```

Grow 2

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify;


    [System.Serializable()]
    public class ThreeTreeTestGrow_1 : TestGene {

        public ThreeTreeTestGrow_1() {
        }

        public virtual object Execute() {
            return this.Sub(c1[0], c1[4]);
        }

        public override object Clone() {
            return new ThreeTreeTestGrow_1();
        }
    }
}
```

Grow 3

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify;
    
    
    [System.Serializable()]
    public class ThreeTreeTestGrow_2 : TestGene {
        
        public ThreeTreeTestGrow_2() {
        }
        
        public virtual object Execute() {
            return this.Mul(c1[5], c1[2]);
        }
        
        public override object Clone() {
            return new ThreeTreeTestGrow_2();
        }
    }
}
```

Grow 4

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify;
    
    
    [System.Serializable()]
    public class ThreeTreeTestGrow_3 : TestGene {
        
        public ThreeTreeTestGrow_3() {
        }
        
        public virtual object Execute() {
            return this.Add(Y, Y);
        }
        
        public override object Clone() {
            return new ThreeTreeTestGrow_3();
        }
    }
}
```

RampedHalfAndHalf 1

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify_;


    [System.Serializable()]
    public class ThreeTreeTestRampedHalfAndHalf_0 : CommandTestGene {

        public ThreeTreeTestRampedHalfAndHalf_0() {
        }

        public virtual void Execute() {
            if (this.E()) {
                this.B();
            }
            else {
                this.A();
                this.A();
                this.D();
            }
        }

        public override object Clone() {
            return new ThreeTreeTestRampedHalfAndHalf_0();
        }
    }
}
```

RampedHalfAndHalf 2

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify_;


    [System.Serializable()]
    public class ThreeTreeTestRampedHalfAndHalf_1 : CommandTestGene {

        public ThreeTreeTestRampedHalfAndHalf_1() {
        }

        public virtual void Execute() {
            if (this.F(X, this.E())) {
                this.B();
                this.D();
                this.A();
            }
            else {
                this.D();
                this.B();
                this.D();
            }
        }

        public override object Clone() {
            return new ThreeTreeTestRampedHalfAndHalf_1();
        }
    }
}
```

RampedHalfAndHalf 3

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify_;


    [System.Serializable()]
    public class ThreeTreeTestRampedHalfAndHalf_2 : CommandTestGene {

        public ThreeTreeTestRampedHalfAndHalf_2() {
        }

        public virtual void Execute() {
            if (Y) {
                if (X) {
                    this.B();
                }
                else {
                    this.B();
                }
            }
            else {
                if (X) {
                    this.A();
                }
                else {
                    this.A();
                }
            }
        }

        public override object Clone() {
            return new ThreeTreeTestRampedHalfAndHalf_2();
        }
    }
}
```

RampedHalfAndHalf 4

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
```

```
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ThreeTreeTest_output {
    using System;
    using Flatland.GP.TestProblems_Do_not_modify_;


    [System.Serializable()]
    public class ThreeTreeTestRampedHalfAndHalf_3 : CommandTestGene {

        public ThreeTreeTestRampedHalfAndHalf_3() {
        }

        public virtual void Execute() {
            this.A();
            this.D();
            this.A();
            this.D();
            this.A();
            this.A();
            this.B();
            this.B();
            this.D();
        }

        public override object Clone() {
            return new ThreeTreeTestRampedHalfAndHalf_3();
        }
    }
}
```

Appendix A3

Test output from crossovertest in CrossoverTest.cs
Crossover point
All crossed nodes

TestCrossover : Passed

```
Crossover: ONE_POINT_CROSS_OVER Initialization: Full
Using example genes Flatland.GP.TestProblems_Do_not_modify.TestGene and Flatland.GP.TestProblems_Do_not_mo
dify.TestGene3
BEFORE tree cross over:
Tree 1:
Depth: 1 - Node  Int32 Div(Int32, Int32)Child count: 2 HashCode: 54636159
Depth: 2 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 42708074
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 42815147
Depth: 3 - Leaf ARRAY_INDEX: 4 HashCode: 63130991
Depth: 3 - Leaf ARRAY_INDEX: 0 HashCode: 22322349
Depth: 3 - Leaf VARIABLE: X HashCode: 64981649
Depth: 3 - Leaf ARRAY_INDEX: 6 HashCode: 38493088
Tree 2:
Depth: 1 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 54467399
Depth: 2 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 35113868
```

```
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 36620214
Depth: 3 - Leaf VARIABLE: Black HashCode: 37296927
Depth: 3 - Leaf VARIABLE: Black HashCode: 640117
Depth: 3 - Leaf VARIABLE: Black HashCode: 28805302
Depth: 3 - Leaf VARIABLE: White HashCode: 21170186
AFTER Tree cross over:
Depth: 1 - Node  Int32 Div(Int32, Int32)Child count: 2 HashCode: 54636159
Depth: 2 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 42708074
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 42815147
Depth: 3 - Leaf VARIABLE: Black HashCode: 640117
Depth: 3 - Leaf ARRAY_INDEX: 0 HashCode: 22322349
Depth: 3 - Leaf VARIABLE: X HashCode: 64981649
Depth: 3 - Leaf ARRAY_INDEX: 6 HashCode: 38493088
Tree 2:
Depth: 1 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 54467399
Depth: 2 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 35113868
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 36620214
Depth: 3 - Leaf VARIABLE: Black HashCode: 37296927
Depth: 3 - Leaf ARRAY_INDEX: 4 HashCode: 63130991
Depth: 3 - Leaf VARIABLE: Black HashCode: 28805302
Depth: 3 - Leaf VARIABLE: White HashCode: 21170186


**************************


AFTER Tree cross over:
Depth: 1 - Node  Int32 Div(Int32, Int32)Child count: 2 HashCode: 54636159
Depth: 2 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 35113868
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 42815147
Depth: 3 - Leaf VARIABLE: Black HashCode: 37296927
Depth: 3 - Leaf ARRAY_INDEX: 4 HashCode: 63130991
Depth: 3 - Leaf VARIABLE: X HashCode: 64981649
Depth: 3 - Leaf ARRAY_INDEX: 6 HashCode: 38493088
Tree 2:
Depth: 1 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 54467399
Depth: 2 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 42708074
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 36620214
Depth: 3 - Leaf VARIABLE: Black HashCode: 640117
Depth: 3 - Leaf ARRAY_INDEX: 0 HashCode: 22322349
Depth: 3 - Leaf VARIABLE: Black HashCode: 28805302
Depth: 3 - Leaf VARIABLE: White HashCode: 21170186


**************************

AFTER Tree cross over:
Depth: 1 - Node  Int32 Div(Int32, Int32)Child count: 2 HashCode: 54636159
Depth: 2 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 42708074
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 42815147
Depth: 3 - Leaf VARIABLE: Black HashCode: 640117
Depth: 3 - Leaf ARRAY_INDEX: 0 HashCode: 22322349
Depth: 3 - Leaf VARIABLE: X HashCode: 64981649
Depth: 3 - Leaf ARRAY_INDEX: 6 HashCode: 38493088
Tree 2:
```

```
Depth: 1 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 54467399
Depth: 2 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 35113868
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 36620214
Depth: 3 - Leaf VARIABLE: Black HashCode: 37296927
Depth: 3 - Leaf ARRAY_INDEX: 4 HashCode: 63130991
Depth: 3 - Leaf VARIABLE: Black HashCode: 28805302
Depth: 3 - Leaf VARIABLE: White HashCode: 21170186


*************************

Crossover: ONE_POINT_CROSS_OVER Initialization: Grow
Using example genes Flatland.GP.TestProblems_Do_not_modify.TestGene and Flatland.GP.TestProblems_Do_not_mo
dify.TestGene3
BEFORE tree cross over:
Tree 1:
Depth: 1 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 13134304
Depth: 2 - Leaf ARRAY_INDEX: 8 HashCode: 54172779
Depth: 2 - Leaf ARRAY_INDEX: 4 HashCode: 21855962
Tree 2:
Depth: 1 - Node  Int32 Green(Int32, Int32)Child count: 2 HashCode: 43994237
Depth: 2 - Leaf VARIABLE: White HashCode: 33583636
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 34868631
Depth: 3 - Leaf VARIABLE: White HashCode: 25584554
Depth: 3 - Leaf VARIABLE: White HashCode: 10454271
AFTER Tree cross over:
Depth: 1 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 13134304
Depth: 2 - Leaf VARIABLE: White HashCode: 33583636
Depth: 2 - Leaf ARRAY_INDEX: 4 HashCode: 21855962
Tree 2:
Depth: 1 - Node  Int32 Green(Int32, Int32)Child count: 2 HashCode: 43994237
Depth: 2 - Leaf ARRAY_INDEX: 8 HashCode: 54172779
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 34868631
Depth: 3 - Leaf VARIABLE: White HashCode: 25584554
Depth: 3 - Leaf VARIABLE: White HashCode: 10454271


*************************

BEFORE tree cross over:
Tree 1:
Depth: 1 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 13134304
Depth: 2 - Leaf VARIABLE: White HashCode: 33583636
Depth: 2 - Leaf ARRAY_INDEX: 4 HashCode: 21855962
Tree 2:
Depth: 1 - Node  Int32 Green(Int32, Int32)Child count: 2 HashCode: 43994237
Depth: 2 - Leaf ARRAY_INDEX: 8 HashCode: 54172779
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 34868631
Depth: 3 - Leaf VARIABLE: White HashCode: 25584554
Depth: 3 - Leaf VARIABLE: White HashCode: 10454271
AFTER Tree cross over:
Depth: 1 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 13134304
Depth: 2 - Leaf ARRAY_INDEX: 8 HashCode: 54172779
Depth: 2 - Leaf ARRAY_INDEX: 4 HashCode: 21855962
Tree 2:
```

```
Depth: 1 - Node  Int32 Green(Int32, Int32)Child count: 2 HashCode: 43994237
Depth: 2 - Leaf VARIABLE: White HashCode: 33583636
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 34868631
Depth: 3 - Leaf VARIABLE: White HashCode: 25584554
Depth: 3 - Leaf VARIABLE: White HashCode: 10454271


**************************

BEFORE tree cross over:
Tree 1:
Depth: 1 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 13134304
Depth: 2 - Leaf ARRAY_INDEX: 8 HashCode: 54172779
Depth: 2 - Leaf ARRAY_INDEX: 4 HashCode: 21855962
Tree 2:
Depth: 1 - Node  Int32 Green(Int32, Int32)Child count: 2 HashCode: 43994237
Depth: 2 - Leaf VARIABLE: White HashCode: 33583636
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 34868631
Depth: 3 - Leaf VARIABLE: White HashCode: 25584554
Depth: 3 - Leaf VARIABLE: White HashCode: 10454271
AFTER Tree cross over:
Depth: 1 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 13134304
Depth: 2 - Leaf VARIABLE: White HashCode: 33583636
Depth: 2 - Leaf ARRAY_INDEX: 4 HashCode: 21855962
Tree 2:
Depth: 1 - Node  Int32 Green(Int32, Int32)Child count: 2 HashCode: 43994237
Depth: 2 - Leaf ARRAY_INDEX: 8 HashCode: 54172779
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 34868631
Depth: 3 - Leaf VARIABLE: White HashCode: 25584554
Depth: 3 - Leaf VARIABLE: White HashCode: 10454271


**************************

Crossover: ONE_POINT_CROSS_OVER Initialization: RampedHalfAndHalf
Using example genes Flatland.GP.TestProblems_Do_not_modify.TestGene and Flatland.GP.TestProblems_Do_not_mo
dify.TestGene3
BEFORE tree cross over:
Tree 1:
Depth: 1 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 680171
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 30607723
Depth: 2 - Node  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 35170261
Depth: 3 - Leaf VARIABLE: X HashCode: 39157888
Depth: 3 - Leaf ARRAY_INDEX: 1 HashCode: 17274517
Depth: 3 - Leaf ARRAY_INDEX: 8 HashCode: 39155797
Depth: 3 - Leaf ARRAY_INDEX: 2 HashCode: 17180427
Tree 2:
Depth: 1 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 34921712
Depth: 2 - Node  Int32 Orange(Int32, Int32)Child count: 2 HashCode: 27973187
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 50833863
Depth: 3 - Leaf VARIABLE: Black HashCode: 5822459
Depth: 3 - Leaf VARIABLE: White HashCode: 60684095
Depth: 3 - Leaf VARIABLE: White HashCode: 46429731
Depth: 3 - Leaf VARIABLE: Black HashCode: 8963119
AFTER Tree cross over:
```

```
Depth: 1 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 680171
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 30607723
Depth: 2 - Node  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 35170261
Depth: 3 - Leaf VARIABLE: White HashCode: 60684095
Depth: 3 - Leaf ARRAY_INDEX: 1 HashCode: 17274517
Depth: 3 - Leaf ARRAY_INDEX: 8 HashCode: 39155797
Depth: 3 - Leaf ARRAY_INDEX: 2 HashCode: 17180427
Tree 2:
Depth: 1 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 34921712
Depth: 2 - Node  Int32 Orange(Int32, Int32)Child count: 2 HashCode: 27973187
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 50833863
Depth: 3 - Leaf VARIABLE: Black HashCode: 5822459
Depth: 3 - Leaf VARIABLE: X HashCode: 39157888
Depth: 3 - Leaf VARIABLE: White HashCode: 46429731
Depth: 3 - Leaf VARIABLE: Black HashCode: 8963119


**************************

BEFORE tree cross over:
Tree 1:
Depth: 1 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 680171
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 30607723
Depth: 2 - Node  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 35170261
Depth: 3 - Leaf VARIABLE: White HashCode: 60684095
Depth: 3 - Leaf ARRAY_INDEX: 1 HashCode: 17274517
Depth: 3 - Leaf ARRAY_INDEX: 8 HashCode: 39155797
Depth: 3 - Leaf ARRAY_INDEX: 2 HashCode: 17180427
Tree 2:
Depth: 1 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 34921712
Depth: 2 - Node  Int32 Orange(Int32, Int32)Child count: 2 HashCode: 27973187
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 50833863
Depth: 3 - Leaf VARIABLE: Black HashCode: 5822459
Depth: 3 - Leaf VARIABLE: X HashCode: 39157888
Depth: 3 - Leaf VARIABLE: White HashCode: 46429731
Depth: 3 - Leaf VARIABLE: Black HashCode: 8963119
AFTER Tree cross over:
Depth: 1 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 680171
Depth: 2 - Node  Int32 Orange(Int32, Int32)Child count: 2 HashCode: 27973187
Depth: 2 - Node  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 35170261
Depth: 3 - Leaf VARIABLE: Black HashCode: 5822459
Depth: 3 - Leaf VARIABLE: X HashCode: 39157888
Depth: 3 - Leaf ARRAY_INDEX: 8 HashCode: 39155797
Depth: 3 - Leaf ARRAY_INDEX: 2 HashCode: 17180427
Tree 2:
Depth: 1 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 34921712
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 30607723
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 50833863
Depth: 3 - Leaf VARIABLE: White HashCode: 60684095
Depth: 3 - Leaf ARRAY_INDEX: 1 HashCode: 17274517
Depth: 3 - Leaf VARIABLE: White HashCode: 46429731
Depth: 3 - Leaf VARIABLE: Black HashCode: 8963119


**************************
```

```
BEFORE tree cross over:
Tree 1:
Depth: 1 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 680171
Depth: 2 - Node  Int32 Orange(Int32, Int32)Child count: 2 HashCode: 27973187
Depth: 2 - Node  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 35170261
Depth: 3 - Leaf VARIABLE: Black HashCode: 5822459
Depth: 3 - Leaf VARIABLE: X HashCode: 39157888
Depth: 3 - Leaf ARRAY_INDEX: 8 HashCode: 39155797
Depth: 3 - Leaf ARRAY_INDEX: 2 HashCode: 17180427
Tree 2:
Depth: 1 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 34921712
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 30607723
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 50833863
Depth: 3 - Leaf VARIABLE: White HashCode: 60684095
Depth: 3 - Leaf ARRAY_INDEX: 1 HashCode: 17274517
Depth: 3 - Leaf VARIABLE: White HashCode: 46429731
Depth: 3 - Leaf VARIABLE: Black HashCode: 8963119
AFTER Tree cross over:
Depth: 1 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 680171
Depth: 2 - Node  Int32 Mod(Int32, Int32)Child count: 2 HashCode: 30607723
Depth: 2 - Node  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 35170261
Depth: 3 - Leaf VARIABLE: White HashCode: 60684095
Depth: 3 - Leaf ARRAY_INDEX: 1 HashCode: 17274517
Depth: 3 - Leaf ARRAY_INDEX: 8 HashCode: 39155797
Depth: 3 - Leaf ARRAY_INDEX: 2 HashCode: 17180427
Tree 2:
Depth: 1 - Node  Int32 Yellow(Int32, Int32)Child count: 2 HashCode: 34921712
Depth: 2 - Node  Int32 Orange(Int32, Int32)Child count: 2 HashCode: 27973187
Depth: 2 - Node  Int32 Blue(Int32, Int32)Child count: 2 HashCode: 50833863
Depth: 3 - Leaf VARIABLE: Black HashCode: 5822459
Depth: 3 - Leaf VARIABLE: X HashCode: 39157888
Depth: 3 - Leaf VARIABLE: White HashCode: 46429731
Depth: 3 - Leaf VARIABLE: Black HashCode: 8963119


**************************
```

Appendix A4

Test output from `MutationsTest in MutationTest.cs`

`Mutation point`

`All mutated nodes`

Node replacement 1

```
EXPRESSION GenoType Flatland.GeneticProgramming.TestProblems_Do_not_modify.Test treedepth 3
Mutation method: Node_Replacement Initialization method: Full
Before Mutation
Tree 1:
Depth: 1 - Node  Int32 Div(Int32, Int32)Child count: 2 HashCode: 35113868
Depth: 2 - Node  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 36620214
Depth: 2 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 37296927
Depth: 3 - Leaf ARRAY_INDEX: 7 HashCode: 640117
Depth: 3 - Leaf ARRAY_INDEX: 7 HashCode: 28805302
```

```
Depth: 3 - Leaf ARRAY_INDEX: 8 HashCode: 21170186
Depth: 3 - Leaf ARRAY_INDEX: 3 HashCode: 13134304
AFTER Mutation:
Depth: 1 - Node  Int32 Div(Int32, Int32)Child count: 2 HashCode: 35113868
Depth: 2 - Node  Int32 Mul(Int32, Int32)Child count: 2 HashCode: 36620214
Depth: 2 - Node  Int32 Add(Int32, Int32)Child count: 2 HashCode: 37296927
Depth: 3 - Leaf ARRAY_INDEX: 7 HashCode: 640117
Depth: 3 - Leaf ARRAY_INDEX: 7 HashCode: 28805302
Depth: 3 - Leaf ARRAY_INDEX: 4 HashCode: 21170186
Depth: 3 - Leaf ARRAY_INDEX: 3 HashCode: 13134304
```

Node replacement 2 – Root node

```
Before Mutation
Tree 1:
Depth: 1 - Leaf ARRAY_INDEX: 4 HashCode: 42708074
Tree 2:
Depth: 1 - Leaf ARRAY_INDEX: 0 HashCode: 42815147
AFTER Mutation:
Depth: 1 - Leaf ARRAY_INDEX: 1 HashCode: 42708074

Tree 2
Depth: 1 - Leaf ARRAY_INDEX: 0 HashCode: 42815147
```

Node replacement 2 – command

```
Before:
Depth: 1 - Node SEQUENCE Child count: 3 HashCode: 64981649
Depth: 2 - Leaf Void A() HashCode: 38493088
Depth: 2 - Leaf Void A() HashCode: 54467399
Depth: 2 - Leaf Void A() HashCode: 35113868
AFTER Mutation:
Depth: 1 - Node SEQUENCE Child count: 3 HashCode: 64981649
Depth: 2 - Leaf Void A() HashCode: 38493088
Depth: 2 - Leaf Void A() HashCode: 54467399
Depth: 2 - Leaf Void B() HashCode: 35113868
```

Subtree mutation: command Function mutated to terminal

```
Tree 1:
Depth: 1 - Node IF_THEN_ELSE Child count: 3 HashCode: 12185796
Depth: 2 - Node  Boolean F(Boolean, Boolean)Child count: 2 HashCode: 11489911
Depth: 2 - Node SEQUENCE Child count: 3 HashCode: 47283970
Depth: 2 - Node SEQUENCE Child count: 3 HashCode: 47403907
Depth: 3 - Leaf VARIABLE: X HashCode: 52801053
```

```
Depth: 3 - Leaf Boolean E() HashCode: 27237168
Depth: 3 - Leaf Void A() HashCode: 17713017
Depth: 3 - Leaf Void A() HashCode: 58888299
Depth: 3 - Leaf Void A() HashCode: 32727789
Depth: 3 - Leaf Void B() HashCode: 63464403
Depth: 3 - Leaf Void B() HashCode: 37325887
Depth: 3 - Leaf Void A() HashCode: 1943350

AFTER Mutation:
Depth: 1 - Node IF_THEN_ELSE Child count: 3 HashCode: 12185796
Depth: 2 - Node  Boolean F(Boolean, Boolean)Child count: 2 HashCode: 11489911
Depth: 2 - Node SEQUENCE Child count: 3 HashCode: 47283970
Depth: 2 - Leaf Void A() HashCode: 50726992
Depth: 3 - Leaf VARIABLE: X HashCode: 52801053
Depth: 3 - Leaf Boolean E() HashCode: 27237168
Depth: 3 - Leaf Void A() HashCode: 17713017
Depth: 3 - Leaf Void A() HashCode: 58888299
Depth: 3 - Leaf Void A() HashCode: 32727789
```

Appendix A5

Run result of MathExample, fitness goal is 74088

Avg fitness history:

```
-2271937, 1541,2, -284318,8, 1653, 935,8, 3813,8, 4423,2, 6127,8, 8917,6, 8853,4,
7555,4, -22106,2, 8913,8, 8993, 8965, 7131,8, 8887,4, 8971,8, 8927, 8945
8970,4, 7566,2, 8970,4, 6815,2, 8950,6, 7182,2, 8924,2, -1269083, 8981,4, 8984,6
-151661,4, 9069,4, 9128,4, 8889,4, 8907,2 ,7123,2, 10248, 8860,6, 8931,8, -204839
-2457136,8, 8540, 8910, 8961, 8982, 7243,6, 7244,2, 7210, 7210, -7573,6
-7556,8, 792, 8978,6, 8944,4, 8978,6, 8531,8, 8910,4, -615162,8, 7567,6, 8333, 6496,2
6462, 8842, 8842,2, 7074,4, 7448,4, 7098,4, 8876,6, 7163,4, 8911,2, 8966,6
9001,2, 8966,8, 9650, -2483957,4, 11047,2, 9656,8, 10321, 11798,6
-50963249,2, 9605,2, 9667,8, 8938, 7900,4, 10110,4, 12537,4, 10791,10769,8
10769,6, 10425,4, 11046,8, 11033,8, 10265,2, 11760,4, 11798,8,9220
-49179886,8, 12147,2, 10723, 10769,6, 10681,6, 10320,2, 8842, -8436
7061,2, 7825,8, 8842,2, 6327, 5637,6, 7074,4, 9203,8, 7800,6, 7788
7818, 8179,2, 6432, 6763,2, -266186980,4, 6763,6, 6618,6, 9632,2, 11093,4
10030,6, 11387,8, 12491,8, 12756,6, 12777,6, 12441,2, 12474,8, 11752,4
10013,2, 9979,8, 12441,4, 12440,8, 12474,4, 12524,6, -2318,6, 12624,8, 12640,8
-3705,4, 12657,8, -7073,2, 12699,8, 11607,8, 10557,6, 10373, 13044,2, 10885,4 ,13178,4
13593,2, -118787, 16738,2, 16679,6, 17418,6, 17502,8, 18981,4, 19133, 20073,4
19737,4, 18712,8, -1224596,4, 23256,6, 22887,2, -1717072,4, 27321,4, -2465
29303,6, -9126243,2, 30294,2, 30294,4, 29949,8, -53462954,6, 36335,6
37765,2, 48350, 2415,4, 59213,6, 60635,6, 59269,4, 58096,6, 58096,2
-26064234,6, 58916,6, 73028,8, 74087, 73020,4, 73020,4, 74087, 59269,4
59622,2, 73717, 59269,6, 74087, 71970,6, 74087, 59269,6, 71970,6
```

72693,2, 58210,6, 44451,2, 56464,4, 71970,6, 71970,6, -357046692,4
71970,6, 74087, 59269,4, 74087, 74079, 73851,6, 59269,8, 71970,8, 57153,2
57153,4, 44099,2, 59269,6, 74087, 58227,8, 59269,6, 74086,8, 74070,4, 74087,2

Appendix A6

Generated code from running MathExample.cs

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace GPRunT07262012220649 {
    using System;
    using Flatland.GP.Examples;


    [System.Serializable()]
    public class Epic_MathExample_Gen_223 : MathExample {

        public Epic_MathExample_Gen_223(System.Runtime.Serialization.SerializationInfo info, 
System.Runtime.Serialization.StreamingContext context) : 
                base(info, context) {
        }

        public Epic_MathExample_Gen_223() {
        }

        public virtual object Execute() {
            return 
this.Sub(this.Mod(this.Sub(this.Mod(this.Div(this.Add(this.Div(this.Sub(this.Mod(this.Sub(X, X), 
this.Mul(X, X)), this.Sub(X, this.Mod(X, X))), this.Mod(this.Sub(this.Sub(this.Div(X, X), 
this.Sub(X, X)), this.Mul(X, X)), this.Sub(this.Sub(X, X), this.Sub(this.Add(X, X), this.Add(X, 
X))))), this.Mod(this.Add(this.Sub(this.Mul(X, X), this.Sub(X, X)), this.Mul(this.Mul(X, X), 
this.Div(X, X))), this.Div(this.Div(X, this.Mod(X, X)), this.Div(this.Mod(X, X), X)))), 
this.Mul(this.Mod(this.Mod(this.Div(this.Mul(X, X), this.Div(X, X)), this.Sub(this.Div(X, X), X)), 
this.Mul(this.Mul(this.Mul(X, X), this.Add(this.Mod(X, X), X)), this.Mul(this.Add(this.Add(X, X), 
X), this.Add(this.Mod(X, X), X)))), this.Sub(this.Div(this.Mod(this.Sub(X, X), this.Mul(X, X)), 
this.Mod(X, X)), this.Add(this.Mul(X, X), this.Mul(X, X))))), 
this.Mul(this.Add(this.Mod(this.Sub(this.Add(this.Mul(this.Div(X, X), this.Mul(X, X)), 
this.Mod(this.Sub(X, X), X)), this.Div(this.Add(this.Div(this.Div(X, X), X), this.Mul(X, this.Sub(X, 
X))), this.Mul(this.Mul(X, X), this.Div(X, X)))), this.Add(this.Div(this.Mod(this.Sub(X, X), 
this.Mul(X, X)), this.Mod(X, this.Sub(X, X))), this.Add(this.Add(X, X), this.Mod(this.Mod(X, X), 
this.Add(X, X))))), this.Mul(this.Sub(this.Sub(this.Sub(X, X), this.Mul(X, X)), 
this.Mul(this.Mod(this.Sub(this.Mul(X, X), X), this.Sub(X, X)), this.Sub(this.Mul(X, X), this.Div(X, 
X)))), this.Add(this.Sub(this.Mod(this.Div(X, X), this.Add(X, X)), this.Mod(this.Sub(this.Add(X, X), 
this.Mul(X, X)), this.Mod(X, X))), this.Mod(this.Add(this.Mod(X, X), this.Div(X, X)), this.Mod(X, 
X))))), this.Mod(this.Add(this.Mod(this.Add(this.Add(X, X), this.Mul(X, this.Sub(X, X))), 
this.Mul(this.Add(X, X), this.Add(this.Div(X, X), X))), 
this.Add(this.Mod(this.Mul(this.Mul(this.Add(X, X), this.Div(X, X)), this.Mod(X, this.Div(X, X))), 
this.Sub(this.Sub(X, X), this.Add(X, X))), this.Add(this.Add(this.Mod(this.Add(X, this.Mul(X, X)), 
this.Mod(this.Div(X, X), X)), this.Sub(this.Mul(X, X), this.Div(X, X))), this.Div(this.Add(X, X), 
this.Add(X, X))))), this.Add(this.Mul(this.Mul(this.Sub(X, X), this.Mod(this.Sub(X, X), this.Mul(X, 
X))), this.Mul(this.Div(this.Mul(X, this.Mod(X, X)), this.Div(X, X)), this.Mul(this.Add(X, 
this.Sub(X, X)), this.Sub(X, X)))), this.Mod(this.Mod(this.Add(this.Mul(X, X), this.Mul(X, 
this.Mod(X, X))), this.Div(this.Mod(this.Mul(X, X), this.Sub(X, X)), this.Add(X, X))), 
this.Mod(this.Mod(X, X), this.Sub(this.Add(X, X), this.Sub(X, X)))))))), 
this.Add(this.Mul(this.Add(this.Sub(this.Mod(this.Mul(this.Mod(this.Mod(X, X), X), this.Sub(X, X)), 
this.Mul(this.Sub(this.Sub(X, X), this.Div(X, X)), this.Mul(X, X))), 
this.Sub(this.Mod(this.Add(this.Mod(X, X), this.Div(X, X)), this.Add(this.Mul(X, X), X)),
this.Div(this.Mod(X, this.Add(X, X)), this.Mul(X, X)))),
this.Add(this.Mul(this.Sub(this.Sub(this.Mod(X, this.Mod(X, X)), this.Sub(X, X)),
this.Add(this.Mod(this.Sub(this.Sub(X, X), X), this.Sub(X, X)), this.Sub(this.Add(X, X), X))),
this.Add(this.Sub(this.Add(X, this.Mod(X, X)), this.Sub(X, X)), this.Add(this.Sub(X, X), this.Add(X,
X)))), this.Mod(this.Add(this.Mod(X, X), this.Mul(X, this.Mod(X, X))),
this.Div(this.Add(this.Sub(this.Sub(X, X), this.Mul(X, X)), this.Add(X, X)),
this.Sub(this.Mod(this.Mul(X, X), X), this.Mod(X, this.Add(X, X))))))),
this.Div(this.Add(this.Mod(this.Mul(this.Sub(this.Div(X, X), this.Add(this.Add(X, X), X)),
this.Sub(this.Add(X, X), this.Mod(this.Mod(X, X), this.Mod(X, X)))),
this.Add(this.Mul(this.Add(this.Mod(X, X), X), this.Sub(X, X)), this.Mod(this.Mod(this.Sub(X, X),
this.Mod(X, X)), this.Mod(this.Mod(X, X), X)))),
this.Add(this.Mul(this.Add(this.Mul(this.Mul(this.Div(X, X), this.Div(X, X)), this.Mod(this.Mul(X,
X), this.Div(X, X))), this.Mul(this.Mod(X, X), this.Mod(this.Add(X, X), X))), this.Div(this.Div(X,
this.Div(X, X)), this.Div(this.Sub(X, X), this.Div(X, X)))), this.Add(this.Add(this.Div(this.Sub(X,
X), X), this.Div(X, X)), this.Add(this.Mul(X, X), this.Add(X, X))))),
this.Add(this.Div(this.Sub(this.Div(this.Div(X, X), X), this.Div(this.Div(this.Add(X, X),
this.Mul(X, X)), this.Add(this.Div(X, X), X))), this.Div(this.Sub(this.Mod(this.Mul(X, X), X),
this.Mod(this.Sub(this.Add(X, X), X), this.Add(X, X))), this.Sub(this.Mod(X, X), this.Sub(X, X)))),
this.Mul(this.Div(this.Mul(this.Mul(X, X), this.Sub(X, X)), this.Div(this.Mod(X, X), X)),
this.Div(this.Sub(X, X), this.Add(this.Mod(X, X), this.Div(X, X))))))),
this.Div(this.Div(this.Mul(this.Sub(this.Sub(this.Mul(X, X), this.Add(X, this.Add(X, X))),
this.Div(this.Div(this.Mul(X, X), this.Sub(this.Mul(X, X), X)), this.Add(this.Div(X, X), this.Mul(X,
X)))), this.Add(this.Mod(this.Mul(this.Sub(X, X), this.Mod(X, X)), this.Sub(this.Sub(X, X),
this.Sub(X, X))), this.Mul(this.Sub(this.Div(X, X), this.Mod(this.Sub(X, X), this.Sub(X, X))),
this.Mul(this.Div(X, this.Mul(X, X)), this.Div(X, X))))), this.Mod(this.Div(this.Sub(this.Div(X, X),
this.Mul(X, X)), this.Mul(this.Mod(X, X), this.Mod(this.Mul(X, X), this.Mul(X, X)))),
this.Add(this.Mod(this.Div(this.Sub(X, X), this.Sub(X, X)), this.Sub(this.Add(X, X), this.Mod(X,
X))), this.Sub(this.Mul(this.Sub(this.Mul(X, X), this.Sub(X, X)), this.Mul(this.Div(X, X),
this.Sub(this.Sub(X, X), this.Div(X, X)))), this.Sub(this.Sub(this.Mul(X, X), X),
this.Mul(this.Div(X, X), this.Mod(X, X))))))),
this.Mul(this.Add(this.Sub(this.Mod(this.Mul(this.Mod(this.Mod(X, X), X), this.Sub(X, X)),
this.Mul(this.Sub(this.Sub(X, X), this.Div(X, X)), this.Mul(X, X))),
this.Sub(this.Mod(this.Add(this.Mod(X, X), this.Div(X, X)), this.Add(this.Mul(X, X), X)),
this.Div(this.Mod(X, this.Add(X, X)), this.Mul(X, X)))),
this.Add(this.Mul(this.Sub(this.Sub(this.Mod(X, this.Mod(X, X)), this.Sub(X, X)),
this.Add(this.Mod(this.Sub(this.Sub(X, X), X), this.Sub(X, X)), this.Sub(this.Add(X, X), X))),
this.Add(this.Sub(this.Add(X, this.Mod(X, X)), this.Sub(X, X)), this.Add(this.Sub(X, X), this.Add(X,
X)))), this.Mod(this.Add(this.Mod(X, X), this.Mul(X, this.Mod(X, X))),
this.Div(this.Add(this.Sub(this.Sub(X, X), this.Mul(X, X)), this.Add(X, X)),
this.Sub(this.Mod(this.Mul(X, X), X), this.Mod(X, this.Add(X, X))))))),
this.Div(this.Add(this.Mod(this.Mul(this.Sub(this.Div(X, X), this.Add(this.Add(X, X), X)),
this.Sub(this.Add(X, X), this.Mod(this.Mod(X, X), this.Mod(X, X)))),
this.Add(this.Mul(this.Add(this.Mod(X, X), X), this.Sub(X, X)), this.Mod(this.Mod(this.Sub(X, X),
this.Mod(X, X)), this.Mod(this.Mod(X, X), X)))),
this.Add(this.Mul(this.Add(this.Mul(this.Mul(this.Div(X, X), this.Div(X, X)), this.Mod(this.Mul(X,
X), this.Div(X, X))), this.Mul(this.Mod(X, X), this.Mod(this.Add(X, X), X))), this.Div(this.Div(X,
this.Div(X, X)), this.Div(this.Sub(X, X), this.Div(X, X)))), this.Add(this.Add(this.Div(this.Sub(X,
X), X), this.Div(X, X)), this.Add(this.Mul(X, X), this.Add(X, X))))),
this.Add(this.Div(this.Sub(this.Div(this.Div(X, X), X), this.Div(this.Div(this.Add(X, X),
this.Mul(X, X)), this.Add(this.Div(X, X), X))), this.Div(this.Sub(this.Mod(this.Mul(X, X), X),
this.Mod(this.Sub(this.Add(X, X), X), this.Add(X, X))), this.Sub(this.Mod(X, X), this.Sub(X, X)))),
this.Mul(this.Div(this.Mul(this.Mul(X, X), this.Sub(X, X)), this.Div(this.Mod(X, X), X)),
this.Div(this.Sub(X, X), this.Add(this.Mod(X, X), this.Div(X, X)))))))))),
this.Sub(this.Mod(this.Div(this.Add(this.Div(this.Sub(this.Mod(this.Sub(X, X), this.Mul(X, X)),
this.Sub(X, this.Mod(X, X))), this.Mod(this.Sub(this.Sub(this.Div(X, X), this.Sub(X, X)),
this.Mul(X, X)), this.Sub(this.Sub(X, X), this.Sub(this.Add(X, X), this.Add(X, X))))),
this.Mod(this.Add(this.Sub(this.Mul(X, X), this.Sub(X, X)), this.Mul(this.Mul(X, X), this.Div(X,
X))), this.Div(this.Div(X, this.Mod(X, X)), this.Div(this.Mod(X, X), X)))),
this.Mul(this.Mod(this.Mod(this.Div(this.Mul(X, X), this.Div(X, X)), this.Sub(this.Div(X, X), X)),
this.Mul(this.Mul(this.Mul(X, X), this.Add(this.Mod(X, X), X)), this.Mul(this.Add(this.Add(X, X),
X), this.Add(this.Mod(X, X), X)))), this.Sub(this.Div(this.Mod(this.Sub(X, X), this.Mul(X, X)),
this.Mod(X, X)), this.Add(this.Mul(X, X), this.Mul(X, X))))),
this.Mul(this.Add(this.Mod(this.Sub(this.Add(this.Mul(this.Div(X, X), this.Mul(X, X)),
this.Mod(this.Sub(X, X), X)), this.Div(this.Add(this.Div(this.Div(X, X), X), this.Mul(X, this.Sub(X,
X))), this.Mul(this.Mul(X, X), this.Div(X, X)))), this.Add(this.Div(this.Mod(this.Sub(X, X),
this.Mul(X, X)), this.Mod(X, this.Sub(X, X))), this.Add(this.Add(X, X), this.Mod(this.Mod(X, X),
this.Add(X, X))))), this.Mul(this.Sub(this.Sub(this.Sub(X, X), this.Mul(X, X)),
this.Mul(this.Mod(this.Sub(this.Mul(X, X), X), this.Sub(X, X)), this.Sub(this.Mul(X, X), this.Div(X,
X)))), this.Add(this.Sub(this.Mod(this.Div(X, X), this.Add(X, X)), this.Mod(this.Sub(this.Add(X, X),
this.Mul(X, X)), this.Mod(X, X))), this.Mod(this.Add(this.Mod(X, X), this.Div(X, X)), this.Mod(X,
X))))), this.Mod(this.Add(this.Mod(this.Add(this.Add(X, X), this.Mul(X, this.Sub(X, X))),
this.Mul(this.Add(X, X), this.Add(this.Div(X, X), X))),
this.Add(this.Mod(this.Mul(this.Mul(this.Add(X, X), this.Div(X, X)), this.Mod(X, this.Div(X, X))),
this.Sub(this.Sub(X, X), this.Add(X, X))), this.Add(this.Add(this.Mod(this.Add(X, this.Mul(X, X)),
this.Mod(this.Div(X, X), X)), this.Sub(this.Mul(X, X), this.Div(X, X))), this.Div(this.Add(X, X),
this.Add(X, X))))), this.Add(this.Mul(this.Mul(this.Sub(X, X), this.Mod(this.Sub(X, X), this.Mul(X,
X))), this.Mul(this.Div(this.Mul(X, this.Mod(X, X)), this.Div(X, X)), this.Mul(this.Add(X,
this.Sub(X, X)), this.Sub(X, X)))), this.Mod(this.Mod(this.Add(this.Mul(X, X), this.Mul(X,
this.Mod(X, X))), this.Div(this.Mod(this.Mul(X, X), this.Sub(X, X)), this.Add(X, X))),
this.Mod(this.Mod(X, X), this.Sub(this.Add(X, X), this.Sub(X, X)))))))),
this.Add(this.Mod(this.Mod(this.Add(this.Add(this.Mod(this.Mod(this.Div(X, X), this.Div(X, X)),
this.Div(this.Sub(X, X), this.Mul(X, X))), this.Div(this.Add(this.Mul(X, X), this.Div(X, X)),
this.Sub(this.Sub(X, X), this.Sub(X, X)))), this.Mod(this.Mul(this.Div(this.Mul(X, this.Add(X, X)),
this.Mul(X, X)), this.Mul(this.Mod(X, X), this.Add(X, X))), this.Div(this.Mod(this.Div(X,
this.Sub(X, X)), this.Div(X, X)), this.Mod(this.Sub(X, X), this.Mul(this.Div(X, X), this.Sub(X,
X)))))), this.Sub(this.Sub(this.Div(X, this.Div(X, X)), this.Div(X, X)),
this.Div(this.Sub(this.Add(X, X), this.Div(X, X)), this.Mul(X, this.Add(X, X))))),
this.Mul(this.Mul(this.Div(this.Mod(this.Add(this.Sub(X, X), this.Add(this.Div(X, X), X)),
this.Div(this.Add(X, X), this.Mod(this.Mul(this.Mul(X, X), X), this.Mod(X, X)))),
this.Mod(this.Mod(this.Div(this.Mul(X, X), this.Mod(X, X)), this.Mul(X, X)), this.Mod(this.Div(X,
X), this.Mod(X, X)))), this.Mul(this.Sub(this.Sub(this.Div(X, X), X), this.Add(X, X)),
this.Div(this.Sub(X, this.Mul(X, X)), this.Sub(this.Div(X, X), this.Sub(X, X))))),
this.Sub(this.Sub(this.Mul(this.Sub(this.Add(X, this.Sub(X, X)), this.Mul(this.Div(this.Div(X, X),
this.Mul(X, X)), this.Mod(X, X))), this.Sub(this.Mul(this.Div(X, X), this.Mul(X, X)),
this.Sub(this.Add(X, X), X))), this.Sub(this.Mod(this.Add(X, X), this.Mul(X, X)),
this.Mul(this.Mul(X, X), this.Mul(X, this.Add(X, X))))), this.Sub(this.Sub(this.Add(this.Mul(X,
this.Mod(X, X)), this.Div(X, this.Mul(X, X))), this.Div(this.Div(this.Mod(X, X), this.Add(X, X)),
this.Mul(this.Mod(X, X), this.Mod(X, X)))), this.Sub(this.Add(this.Mul(this.Add(X, X), X),
this.Sub(X, X)), this.Add(this.Div(this.Sub(X, X), X), this.Mod(X, X))))))),
this.Div(this.Add(this.Mul(this.Div(this.Mod(this.Add(X, this.Mod(X, X)), this.Sub(X, X)),
this.Mul(this.Sub(X, X), this.Div(X, X))), this.Sub(this.Mul(this.Mul(this.Sub(X, X), this.Sub(X,
X)), this.Mod(this.Mul(X, X), this.Mul(X, this.Div(X, X)))), this.Mod(this.Div(this.Div(X,
this.Div(X, X)), this.Add(this.Mul(X, X), this.Mod(X, X))), this.Sub(this.Add(X, X),
this.Mul(this.Sub(X, X), X))))), this.Mod(this.Div(this.Sub(this.Mul(X, X), this.Div(this.Add(X, X),
X)), this.Mod(this.Mod(X, X), X)), this.Add(this.Sub(this.Sub(X, X), this.Sub(X, X)),
this.Sub(this.Add(X, X), this.Mul(X, X))))), this.Mod(this.Sub(this.Mod(this.Mod(this.Mul(X, X),
this.Mul(X, this.Div(X, X))), this.Sub(X, this.Add(X, X))), this.Mod(this.Mul(this.Add(X, X),
this.Mod(X, X)), this.Sub(this.Div(X, X), this.Add(this.Mul(X, X), X)))),
this.Mod(this.Mod(this.Mod(this.Mul(this.Mod(X, X), this.Sub(X, X)), this.Div(this.Sub(X, X),
this.Sub(X, X))), this.Mul(this.Sub(this.Sub(X, X), this.Div(X, X)), this.Sub(this.Mod(X, X),
this.Sub(X, X)))), this.Sub(this.Sub(this.Mod(this.Add(X, X), this.Mod(X, X)), this.Sub(this.Add(X,
X), X)), this.Div(this.Add(this.Sub(X, X), X), this.Sub(this.Sub(X, X), this.Sub(this.Mod(X, X),
this.Sub(X, X)))))))))), this.Mod(this.Sub(this.Sub(this.Mod(this.Add(X, X), this.Sub(X, X)),
this.Mul(this.Mod(X, this.Mul(X, X)), this.Mul(X, X))), this.Add(this.Add(X, this.Div(X, X)),
this.Div(this.Mul(X, X), X))), this.Mod(this.Mul(this.Div(this.Div(X, X), this.Add(X, X)),
this.Div(X, X)), this.Mul(this.Mul(X, X), this.Sub(this.Div(X, X), this.Mul(X, X))))));
        }

        public override object Clone() {
            return new Epic_MathExample_Gen_223();
        }
    }
}
```

Appendix B1

```
namespace Flatland.GP.CoreTypes {
    /// <summary>
    /// Genotype for genetic code.
    ///
    /// Convention that implementers of the GenoType must follow.
    /// 1.     The inheriting class must be:
    ///     a. public
    ///     b. serializable
    ///     c. cloneable
    ///     d. exposing a public parameterless constructor.
    ///     e. defining primitives for both the function and the primitive set.
    ///     f. inheriting directly from the GenoType class.
    /// 2.     The primitives must be public.
    /// </summary>
    [Serializable]
    public abstract class GenoType : Behavior, ICloneable {
        /// <summary>
        /// Implementers should return a fitness value indicating the fitness of the generated code.
        /// This value will normally make sense only in the context of a gentic run with a set fitness goal.
        /// </summary>
        /// <returns>A fitness value.</returns>
        public abstract double Fitness();

        /// <summary>
        /// Expression Tree representation of the genetic code
        /// </summary>
        public Tree Tree { get; set; }

        /// <summary>
        /// Reports must be clonable.
        /// </summary>
        /// <returns></returns>
        public virtual object Clone() {
            throw new NotImplementedException("Clone must be implemented");
        }

        /// <summary>
        /// Execute the generated code.
        /// </summary>
        /// <returns></returns>
        public object Execute(){
            return Tree.Execute(this);
        }
    }
}
```

Appendix B2

Template for defining a genotype for the GP

```
namespace Flatland.GP.Examples {
    /// <summary>
    /// Template for a Gene implementation. It shows what member must at least be implemented for the
template to be valid marked with.
    /// Examples can be found in the Examples namespace
    /// </summary>
    [Serializable]
    public class Template : GenoType {

        /*
         * ********* Implement your Primitive. *************
         * [Primitive]
         * public void MoveLeft() {
         *    Agent.SetDirection((float)Math.PI);
         * }
         */

        /// <summary>
        /// A genotype must expose a public parameterless constructor.
        /// </summary>
        public Template() { }

        /// <summary>
        /// Called when this genotype should act for the Agent.
        /// </summary>
        /// <param name="msg">Message from act</param>
        protected override void Act(ref string msg) {
            /************* Execute your the program with the Agent in Flatland *****************
             *
             * Execute();
             *
             */
        }

        /// <summary>
        /// When using GPRun this should return a fitness value.
        /// This is often done by setting variables combined with a call to Execute that executes the genetic
code. </summary>
        /// <returns></returns>
        public override double Fitness() {
            //var fitness = Execute();
            return 42;
        }

        /// <summary>
        /// A genotype is clonable.
        /// TODO: Must be implemented</summary>
        /// <returns></returns>
        public override object Clone()
        {
            throw new NotImplementedException();
        }

        /// <summary>
        /// A genotype must expect to be deserialized and this constructor must be provided.
        /// Consider implementing GetObjectData</summary>
        /// <param name="info"></param>
        /// <param name="context"></param>
        public Template(SerializationInfo info, StreamingContext context) {
        }
    }
}
```

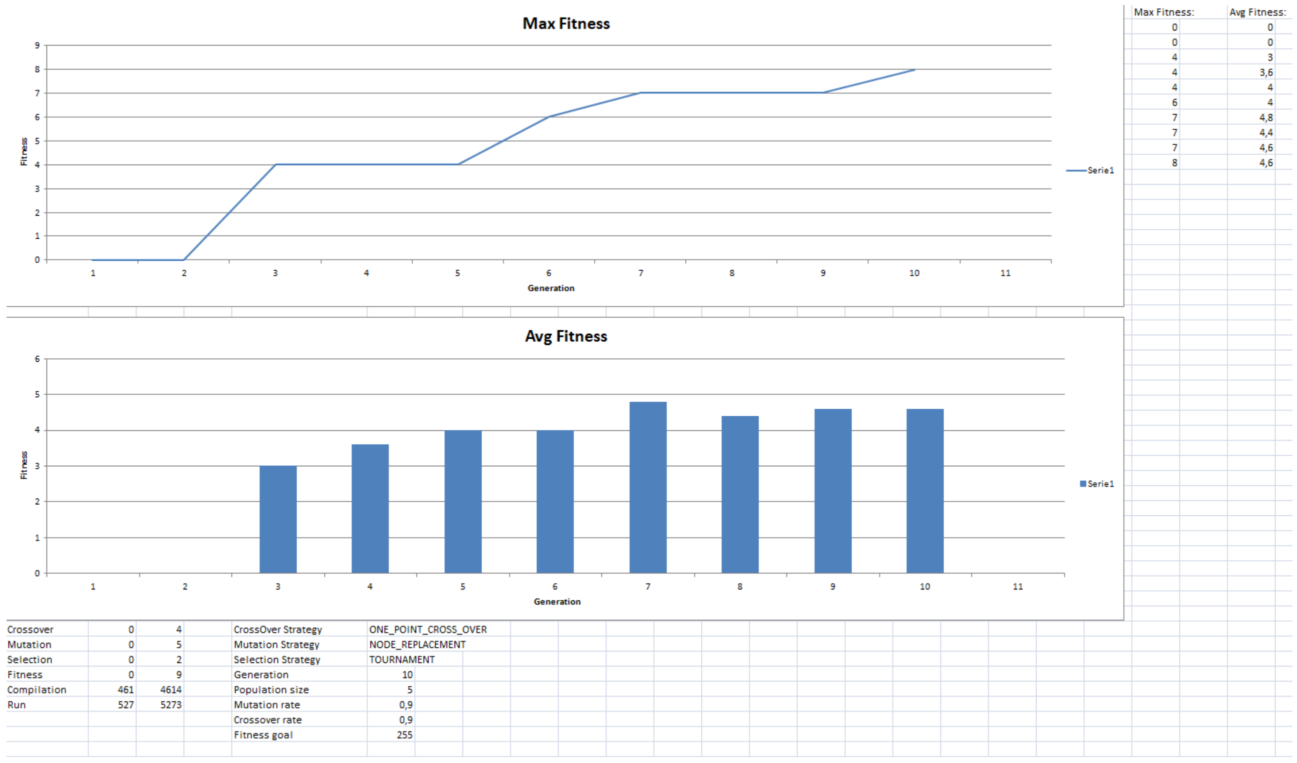
Appendix B3

Adding a new initialization technique

```
    public enum InitializationMethod {
        [GPMethod(typeof(FullInitialization))]
        Full,

        [GPMethod(typeof(GrowInitialization))]
        Grow,

        [GPMethod(typeof(RampedHalfAndHalf))]
        RampedHalfAndHalf

    }
```

Appendix C1

Runresult written to Excel:



Max Fitness:	Avg Fitness:
0	0
0	0
4	3
4	3,6
4	4
6	4
7	4,8
7	4,4
7	4,6
8	4,6

Crossover	0	4	CrossOver Strategy	ONE_POINT_CROSS_OVER
Mutation	0	5	Mutation Strategy	NODE_REPLACEMENT
Selection	0	2	Selection Strategy	TOURNAMENT
Fitness	0	9	Generation	10
Compilation	461	4614	Population size	5
Run	527	5273	Mutation rate	0,9
			Crossover rate	0,9
			Fitness goal	255

Example of a run result of the GP written to text file:

Max fitness: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Avg fitness: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

Description --- Avg. time (ms) --- Total time (ms)
Crossover --- 0 --- 4
Mutation --- 0 --- 4
Selection --- 0 --- 2
Fitness --- 1 --- 14
Run --- 4944 --- 49441

CrossOver Strategy: ONE_POINT_CROSS_OVER
Mutation Strategy: NODE_REPLACEMENT

Selection Strategy: TOURNAMENT
Generation Strategy: 10
Population size: 5
Mutation rate: 0,05
Crossover Strategy: 0,9
Fitness goal: 25

Appendix C2

Find food genotype currently used in Flatland.

```
namespace Flatland.GeneBank {
   /// <summary>
   /// Find food GenoType
   /// </summary>
   [Serializable]
   public class FindFood : GenoType {

      [Primitive]
      public void TurnLeft() {
         Agent.TurnLeft();
      }

      [Primitive]
      public void TurnRight() {
         Agent.TurnRight();
      }

      [Primitive]
      public void Stop() {
         Agent.Stop();
      }

      [Primitive]
      public bool SeesFood() {
         return Agent.SeesFood;
      }

      [Primitive]
      public void Sniff() {
         if (Agent.SeesFood) {
            Agent.Move();
         }
         else {
            Agent.Stop();
            Agent.TurnLeft();
         }
      }
      /// <summary>
      /// Called by the behavior
      /// </summary>
      protected override void Act(ref string msg)
      {
         Execute();
      }

      public override double Fitness()
      {
         return FoodLab.AMOUNT_OF__TEST_FOOD - Math.Abs(FoodLab.AMOUNT_OF__TEST_FOOD - Agent.FoodEaten);
      }

      public override object Clone() {
         return new FindFood();
      }
```

```
    public FindFood(SerializationInfo info, StreamingContext context) { }
    public FindFood() { }
  }
}
```

Appendix C3

Phenotype code of C2

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17379
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace Flatland_129881455965424511.Flatlander_NO_ID {
    using System;
    using Flatland.GeneBank;


    [System.Serializable()]
    public class comp_T07302012201720_FindFood : FindFood {
        
        public comp_T07302012201720_FindFood(System.Runtime.Serialization.SerializationInfo info, 
System.Runtime.Serialization.StreamingContext context) : 
                base(info, context) {
        }
        
        public comp_T07302012201720_FindFood() {
        }
        
        public virtual void Execute() {
            if (this.SeesFood()) {
                if (this.SeesFood()) {
                    this.Sniff();
                    this.TurnLeft();
                    this.Sniff();
                }
                else {
                    this.TurnLeft();
                    this.TurnRight();
                    this.TurnLeft();
                }
            }
            else {
                this.Stop();
                this.Sniff();
                this.Stop();
                if (this.SeesFood()) {
                    this.Sniff();
                }
                else {
                    this.TurnLeft();
                }
                if (this.Move()) {
                    this.Stop();
                }
                else {
                    this.TurnRight();
                }
            }
            if (this.Move()) {
                this.Sniff();
                this.TurnLeft();
                this.TurnLeft();
                this.TurnRight();
                this.Sniff();
                this.Sniff();
                if (this.Move()) {
                    this.Stop();
                }
                else {
                    this.TurnLeft();
                }
            }
            else {
                if (this.SeesFood()) {
                    if (this.Move()) {
                        this.TurnRight();
                    }
                    else {
                        this.TurnLeft();
                    }
                }
                else {
                    if (this.SeesFood()) {
                        this.Stop();
                    }
                    else {
                        this.Stop();
                    }
                }
            }
            this.Sniff();
            this.TurnLeft();
            this.TurnRight();
            this.Stop();
            this.TurnLeft();
            this.TurnRight();
            if (this.SeesFood()) {
                this.TurnLeft();
            }
            else {
                this.Sniff();
            }
            if (this.SeesFood()) {
                this.TurnRight();
            }
            else {
                this.Sniff();
            }
            this.Sniff();
            this.Sniff();
            this.Stop();
            this.TurnRight();
            this.Sniff();
            this.Sniff();
            if (this.SeesFood()) {
                this.Sniff();
            }
            else {
                this.TurnRight();
            }
            if (this.Move()) {
                this.Sniff();
            }
            else {
                this.Stop();
            }
            this.Sniff();
            this.TurnRight();
            this.TurnRight();
        }

        public override object Clone() {
            return new comp_T07302012201720_FindFood();
        }
    }
}
```