Property-Based Testing of Genetic Algorithms

Applying QuickCheck to Validate Core Properties in the Open-Source Library 'GeneticSharp'

Authors:
Christian Skafte Beck Clausen
Jonas Vedsted Sørensen

Contents

1 Introduction

The goal of this project is to use QuickCheck to test a set of properties for a selected open-source software library. The authors of the project have contributed equally to both the report and the code.

1.1 Context

The context of this project is genetic algorithms (GA) which is a sub category of evolutionary algorithms. GAs is based on natural selection where each generation inherits genes from their parent generation via mutation and crossover. A genetic algorithm is initialized with a user defined population and terminates when a certain fitness or time constraint is met [1].

The genetic algorithm domain uses the terms listed in table 1.1. These terms will be used through out the report.

Term	**Definition**
Population	The initial chromosomes used by the algorithm
Generation	Contains possible solutions which is created from the previous generation or the initial population
Solution	A string of numbers, normally zeros and ones. The string represents a possible solution
Mutation	The act of creating a new solution by manipulating a existing solution
Crossover	The act of combining two solutions to produce a new solution
Fitness	A rating calculated for each solution. Higher ratings equals better solutions

Table 1.1: Terms used in the genetic algorithms domain

For this project the library GeneticSharp (GS) was used for property based testing [2]. GS is a genetic algorithm library which comes with build in methods for selection, mutation, crossover, etc. GS is a highly extensible framework capable using a range of default or customized implementations. Almost every concept is abstracted by interfaces, even the GA itself. This means that a user can provide their own implementation of the core (driver) of the GA. The combination of implementations produce GAs with different properties and capabilities. In GS, solutions are known as chromosomes. Furthermore, the GS framework claims to have 100% code coverage from their tests.

1.2 QuickCheck Port

The QuickCheck port used in this project is the .NET variant FsCheck based on the Haskell's QuickCheck. FsCheck allows for property testing in projects using F#, C# and VB. FsCheck comes with generators and shrinkers for basic types such as `int`, `strings`, `lists` and `pairs`, equal to many other ports of QuickCheck. Furthermore, it allows for creating custom generators and shrinkers by providing helper methods such as `map`, `map2`, `growingElements` and `listOf`. FsCheck does not provide statistics for their generators, however it is mentioned in the documentation that the distributions are uniform[3]. FsCheck also provide more advanced functionalities such as model-based testing and support for other testing frameworks from .NET such as xUnit and NUnit[3][4].

1.3 Scope

Since GS is modular and extensible, testing all implementations in the framework would require huge efforts and therefore, a subset of implementations was chosen. GS does not provide a default fitness function and therefore a custom implementation was used. Table 1.2 shows the selected implementations which was selected based on the assumption that these would be simple to suggest properties for. This assumption is backed up by the fact that the best solution is known beforehand. Elite selection is commonly used and was therefore chosen in this project. In the rest of the report, the set of these implementations is referred to as the Integer Maximization Problem (IMP).

Implementation	**Part of GS**	**Description**
EliteSelection	True	Makes sure that the best fit individuals from a generation are always selected for the next generation. Or more formally, ensures that the fitness of the best solution monotonically increases over time[5].
OnePointCrossover	True	The crossover mechanism used. One point crossover means that the solution (string of bits) are split in a single point and the tail is exchanged, resulting in two new children.
FlipBitMutation	True	The mutation mechanism used. Flip bit means that there is a probability that a single bit is flipped. This makes sure that the genetic algorithm does not stall.
IntegerChromosome	False	Represents a 32-bit signed integer as a string of bits.
IntegerMaximizationFitness	False	Returns the value of the IntegerChromosome as fitness. If the integer value is below zero, zero is returned.

Table 1.2: Implementations used for the GA

In addition to the aforementioned implementations, GeneticSharp's template Traveling Salesman Problem (TSP) was subject for testing. The reasoning behind this decision was to test default implementations under the assumption that these make correct use of the GeneticSharp framework. TspChromosome and TspFitness is part of the GS source but not included in the NuGet distribution.

Implementation	**Part of GS**	**Description**
EliteSelection	True	See table 1.2
OrderedCrossover (OX1)	True	Two parents with ordered genes are split up at the same random index and the remaining is switched, resulting in two new children. Ordered crossover ensures that all genes are represented in both children.
TworsMutation	True	Allows the exchange of position of two genes chosen at random. In this case it allows exchanging the position of two cities.
TspChromosome	Extension	Represents an ordered list of cities. The sequence is the order in which each city should be visited to minimize the total distance traveled. Also called the solution path.
TspFitness	Extension	Calculates the mean distance between the cities on the route. The mean value is normalized between 0 and 1 and returned as the fitness.

Table 1.3: Implementations used for TSP

2 Test Design

This section describes the design considerations about the tests. This includes which properties the under test and which input types that must be generated.

2.1 Properties

The properties under test are dependent on the input for the GA, which in this case are the implementations mentioned in subsection 1.3. The literature on GA was briefly explored to find formal properties under consideration. The literature indicates limited knowledge on the formal properties of GA, but a few properties for the crossover function was found[6]. The scope of this project is the use of elite selection and therefore, crossover properties was not considered. In consideration of the limited available properties of GA, we suggest four informal properties as shown in Table 2.1 to verify GA when using an elite selection mechanism.

ID	Informal Property	Description
P1	Termination	The number of generations should match the termination criteria
P2	Monotonically increasing fitness (Elite Selection)	The $n^{th} + 1$ generation's best fitness is greater than or equal to the n^{th} generation's best fitness. The best solution must never be worse that its predecessor.
P3	Equal Fitness	Providing two identical initialized GA's (including a seed) should result in two solutions with Equal Fitness
P4	Equal Best Genes	Providing two identical initialized GA's (including a seed) should result in two solutions with Equal Best Genes

Table 2.1: Informal Properties

2.2 Generators

The GA needs randomized input in order to test its properties. The idea of a GA generator is to take a seed and termination criteria as input and produce an output which can be used for comparison. The seed and termination criteria can be generated by built-in int generators. The output must store necessary information about a GA execution. The GA object produced by GeneticSharp does not provide this information, and therefore, an output object that enhances this was created. The output in this case is an object with the fields listed in Table 2.2.

Field	Description
GA	The GA object produced by the GeneticSharp library.
Termination	The termination criteria that was used for the GA.
Seed	The seed that was used to generate the exact GA execution.
BestChromosomes	A list of the best chromosome (determined by fitness) in each generation. Sorted ascending by generation number.

Table 2.2: Output of a GA execution

Two generators that generates the aforementioned output has been designed. The first generator (GAGenerator) generates a single GA output. The second generator (TwinGAGenerator) takes a GA output as input and generates a tuple of identical initialized GA outputs. The GAGenerator is used to check properties that must hold for a single GA output (**P1** and **P2**). The TwinGAGenerator is used to check properties that must hold for identical initialized GA's outputs (**P3** and **P4**).

3 Implementation

This section showcases the implementations used for the property-based testing of GeneticSharp. It is important to note that return types was added to the functions shown in the code examples. This was done to improve the readability of the code blocks.

The code for the generators and the properties is in this section only shown for the IMP. The reason was that the TSP is only deviating minimally from the IMP.

3.1 Adding Seed to GeneticSharp

The GeneticSharp framework does not have support for configuring the seed before executing the GA. Since a seed is required for running the tests, the GeneticSharp library was extended.

Two classes was created, `FastRandomizationWithSeed` and `FastRandomWithSeed` both which are almost identical to their default implementations[7][8]. Listing 1 shows the addition of a seed property in the constructor. Listing 2 shows the addition of a seed to the default FastRandomization provider used by GeneticSharp. A new method `setSeed()` and local variable `_seed` was added. Listing 3 shows an example of how to initialize the GeneticSharp GA with a seed. Note that the randomization provider is static and therefore all random numbers generated in GeneticSharp depends on the same provider. If all implementations use the static randomization provider to generate random numbers it can be guaranteed that the sequence of pseudo random numbers can be reproduced.

```
1 public FastRandomWithSeed(int seed)
2 {
3     Reinitialise(seed);
4     this.seed = seed;
5 }
6 public int Seed { get; }
```

Listing 1: Seed added to FastRandom

```
1 public static int _seed;
2 public static void setSeed(int seed)
3 {
4     _globalRandom = new FastRandomWithSeed(seed);
5     _threadRandom = new ThreadLocal<FastRandomWithSeed>(NewRandom);
6     _seed = seed;
7 }
```

Listing 2: Seed added to FastRandomization

```
1 // Set the RandomizationProvider in GeneticSharp
2 int seed = 1234;
3 RandomizationProvider.Current = new FastRandomRandomizationWithSeed();
4 FastRandomRandomizationWithSeed.setSeed(seed);
5
6 // .. Initialize and execute the GeneticSharp GA as usual
```

Listing 3: Configuring the seed

3.2 Integer Maximization Genetic Algorithm Generator

To generate the GA, the generator `GAGen` was created. As shown in Listing 4 the generator uses the helper function `map2` (line 10) from FsCheck to create a generator from a function and two generators as inputs to that function. The function used is the `CreateGA` (line 1) which takes a seed and termination criteria as input. The seed and termination is generated using two custom generators returning a integer within a interval (line 4-9). The result of the generator is a `GAResult` object containing the fields of a GA execution as mentioned in subsection 2.2. The `GAResult` implementation can be seen in Appendix B.

```
1  let CreateGA seed termination : GAResult = ...
2
3  let GAGen : Gen<GAResult> =
4      let terminationMin = 0
5      let terminationMax = 1000
6      let terminationGen = Gen.choose (terminationMin,terminationMax)
7      let seedMin = 0
8      let seedMax = Int32.MaxValueGAProperties
9      let seedGen = Gen.choose (seedMin,seedMax)
10     Gen.map2 CreateGA seedGen terminationGen
11 ;;
```

Listing 4: Generator for creating genetic algorithms

3.3 Twin Genetic Algorithms Generator

To generate two GAs from the same seed and termination criteria, the `TwinGAGen` generator was created. The generator makes use of the `GAGen` generator to create the first GA and then reuses the seed and termination to create the twin. This is done by using the `map` function (line 8) creating a generator from a function and a generator. The function is the `CreateTwin` which takes a GAResult as input. The input is generated by the `GAGen` generator.

```
1  let CreateGA seed termination : GAResult = ...
2  let GAGen : Gen<GAResult> = ...
3
4  let TwinGAGen : Gen<GAResult * GAResult> =
5      let CreateTwin (gaRes:GAResult) =
6          let gaRes2 = CreateGA (gaRes.Seed) (gaRes.Termination) in (gaRes, gaRes2)
7      in Gen.map CreateTwin GAGen
8  ;;
```

Listing 5: Generator for creating two genetic algorithms from same seed

3.4 Defining the Tests

The properties defined in subsection 2.1 was implemented in F# (See Listing 6) by creating the type `GAProperties`. By registering a type to the FsCheck `Arb` object, FsCheck is able to test all defined properties (line 17). When the type is registered, FsCheck is able to run the tests the specified amount of times (line 18). In the implementation the quick configuration is used, which only shows when a test has completed all its runs or an error occurs. Furthermore, a custom comparison operator was defined (line 1-4) to compare and print the comparison. This was beneficial to figure out why a tests failed.

```
(* Custom comparison. This compares and print the left and right sides. *)
let (.=.) left right = left = right |@ sprintf "%A = %A" left right ;;
(* Custom comparison of two lists displaying deviations if they occur. *)
let (.==.) left right = ...

type GAProperties =
  static member ``P1 - Number of generations match the termination criteria``
    (gaRes:GAResult) =  gaRes.Termination .=. gaRes.GA.GenerationsNumber
  static member ``P2 - N+1 generation has same or better fitness than N``
    (gaRes:GAResult) = gaRes.BestChromosomes .==. (List.sortBy (fun e -> e.Fitness.
    GetValueOrDefault()) gaRes.BestChromosomes)
  static member ``P3 - Two GA's with the same inputs should result in two solutions with
    the same fitness``
    (g1:GAResult, g2:GAResult) = g1.BestFitness .=. g2.BestFitness
  static member ``P4 - Two GA's with the same inputs should result in two identical best
    genes``
    (g1:GAResult, g2:GAResult) = g1.BestGenes .=. g2.BestGenes
;;

Arb.register<MyGenerators>()
Check.All<GAProperties> ({Config.Quick with MaxTest = 10000})
```

Listing 6: Properties defined

4 Results

This section presents the output of running FSCheck with the implementation presented in section 3. In order to ensure that the properties worked as expected, a set of bugs was introduced in the implementation. The implementation was then run without the bugs. The output for this run is explained and potential bugs are presented for any failing property.

4.1 Introducing bugs

Four bug was introduced in the implementation of the IMP to ensure that FsCheck was able to catch all violated properties. **P1** can be violated by increasing the termination criteria with one. Another way is to provide a termination criteria < 1 which results in the GA terminating after one generation. It was expected that the GA would terminate immediately. However, the intended behavior is not documented in the GS framework and is not considered critical. **P2** can be violated by explicitly inserting the worst possible solution into the `GAResult HookChromosome` list at an index where it is reasonable to assume, that the worst possible solution will never occur. **P3** and **P4** can be violated by increasing the seed by one when generating a twin. This will yield different genes and fitness values of the GA executions and thus violate the properties. The output for violating the aforementioned properties is shown in Appendix C.

4.2 Executing the Tests

This section presents the results for executing the FsCheck tests for both the IMP and the TSP.

Executing the tests for IMP resulted in the output shown Listing 7. The output shows that property **P1**, **P3** and **P4** was passing while **P2** failed. The tests were run 10.000 times which gives a confidence in passing of the properties. Property P2 has on the other hand failed after running one test. The output from **P2** shows that a solution with a lower fitness than previously observed, was considered the best solution. This violation was able to be reproduced multiple times even though only the minimal example is presented here.

```
1  ###################### Integer Maximization ######################
2  P1 - Number of generations match the termination criteria-Ok, passed 10000 tests.
3
4  P2 - N+1 generation has same or better fitness than N-Falsifiable, after 1 test
5  Label of failing property: Fitness decreased at index: 1
6  For generations 1-2:
7   [00000000000000000000000000000111; 00000000000000000000000000000110]
8  Original:
9  {
10     HashCode: 57523360,
11     initial: {
12         Seed: -991515906,
13         termination: 397
14     },
15     result: { termination: 397} }
16
17 P3 - Two GA's with the same inputs should result in two solutions with the same fitness-
       Ok, passed 10000 tests.
18
19 P4 - Two GA's with the same inputs should result in two identical best genes-Ok, passed
       10000 tests.
```

Listing 7: FsCheck output

The tests was executed for the TSP, resulting in property **P1**, **P3** and **P4** passing, while **P2** failed as seen in Appendix D.

4.3 Summary

The combination of the introduction of bugs and the number of tests run for both the IMP and TSP, gives confidence that the property-based tests are functioning. The results from both IMP and TSP indicates that a bug is present in the GA when using elite selection. This has resulted in the creation of a GitHub issue to the developer of GeneticSharp. The issue explains the bug and how to replicate it. At the time of writing a response is not received and therefore the presence of a bug cannot be confirmed by the developer.

The issue can be found on GitHub: `https://github.com/giacomelli/GeneticSharp/issues/72`

5 Discussion

This section discuss the application of property-based testing in the domain of GA.

5.1 Properties

Different approaches was considered for testing properties of a GA. One has to decide on a compromise between speed and variation. For example, a single property can be tested against a single execution of the GA which results in testing on a larger amount of GA executions - one for each property times the amount of tests generated. However, this approach may be slow compared to testing multiple properties for a single GA execution. Speed was not an issue when testing With a limited set of properties and limited run-time of the GA.

Suggesting properties and testing them in the domain of GA is a difficult task. Since GAs belong to the category of search algorithms, testing properties on a GA where the optimal solution was known beforehand (IMP) helped to inspect the output and behavior. When violations of properties occur on simple problems there is a probability that harder problems violate these too. For this reason, the FsCheck was applied to the TSP implementation which indeed indicated a violation of **P2**.

A few pitfalls are present when testing GAs. Firstly, GA is a search algorithm which has the potential for running for extended periods of time. Therefore, it is important to consider how the implementations under test can be simplified. For the TSP it could be to limit the amount of cities and use examples where the optimal route is known. Secondly, GA executions are random and depends on the seed and initial conditions. If an error occurs for a specific seed, the error are likely not to be caught by the tests within a reasonable amount of runs. Having control over the seed to be able to replay the execution is necessary to enable debugging.

5.2 Shrinkers in Genetic Algorithms Domain

Shrinkers is an essential part of the property-based testing domain. However, in this project shrinkers has been deemed unnecessary due to the computation heavy nature of the GAs. This decision was supported further by the few inputs to the GA, thereof the seed and termination criteria. It was deemed unnecessary to shrink the seed of the GA as the quality of the counterexample could decrease. Shrinking of the termination was however still explored in order to examine the results. The shrinker was able to decrease the termination criteria, resulting in the shortest run possible. This was however still computation heavy compared to iterating through the list of best solutions to identify the problem, and the shrinker was therefore inferior. The shrinker used for the experiment can be found in Appendix E.

Shrinker might become necessary if adding new input parameters or properties which can benefit from shrinking.

5.3 Code Coverage

The GS framework reports 100% code coverage on Github. Code coverage is a measure that indicates how many percent of the code that was executed given tests as input. If all lines are executed at least once, then the result is 100%. However, code coverage is not a measure intended for exclusion of bugs. Bugs may still be present in the software, for example because of edge cases or because certain partitions of inputs was bot tested. It is important to notice that the GS framework does not claim to be bug free and therefore, this is not intended as a critique. It is intended that the user of a given library should be aware that unexpected bugs may still occur.

GS's github page contains a list of projects, papers, etc. in which some of them used the EliteSelction mechanism [9][10][11]. The results presented in this report may indicate that some of these projects might contain erroneous or unexpected results.

6 Conclusion & Future Work

Conducting this project resulted in suggesting and testing four properties for two problems of GAs using the GeneticSharp framework. The tests highlighted a potential bug which was violating a suggested property of the elite selection mechanism for IMP and TSP. The bug was filed as an issue to the GitHub repository. The use of shrinkers did not provide any benefits, but is subject for further investigation.

In the future it would be beneficial to begin testing the specific properties for the individual GA components (Crossover, Selections, etc.) as well as different compositions of components. This would provide insight into the correctness of each component as well as the complete GA. Property-based testing and formal validation is not common in the GA domain. GA is normally used to generate input for tests. This opens the opportunity for a new research area. The properties could benefit to be generalized to fit multiple GA domains.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson, 2010. ISBN: 9780136042594.

[2] Diego Giacomelli. *GeneticSharp Github Repository*. May 23, 2020. URL: https://fscheck.github.io/FsCheck/index.html.

[3] fscheck. May 23, 2020. URL: https://fscheck.github.io/FsCheck/index.html.

[4] fscheck. May 23, 2020. URL: https://github.com/fscheck/FsCheck.

[5] Shumeet Baluja and Rich Caruana. "Removing the Genetics from the Standard Genetic Algorithm". In: *Proceedings of the Twelfth International Conference on International Conference on Machine Learning*. ICML'95. Tahoe City, California, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 38–46. ISBN: 1558603778.

[6] Akihiko Uchibori and Noboru Endou. "Basic Properties of Genetic Algorithm". In: *Journal of Formalized Mathematics* 11 (1999), pp. 151–160. URL: http://mizar.uwb.edu.pl/JFM/Vol11/genealg1.html.

[7] Diego Giacomelli. *GeneticSharp FastRandomRandomization Implementation*. May 24, 2020. URL: https://github.com/giacomelli/GeneticSharp/blob/master/src/GeneticSharp.Domain/Randomizations/FastRandomRandomization.cs.

[8] Diego Giacomelli. *GeneticSharp FastRandom Implementation*. May 24, 2020. URL: https://github.com/giacomelli/GeneticSharp/blob/master/src/GeneticSharp.Domain/Randomizations/Externals/FastRandom.cs.

[9] Elvan Kula & Hans Schouten. *AeroVision - Aircraft Trajectories Optimization and Visualization*. May 26, 2020. URL: https://repository.tudelft.nl/islandora/object/uuid:91c8261d-a5f2-414a-9b83-2e0d6ad5b37f/datastream/OBJ.

[10] Ahmet Can Saner. *DESIGN OF A WAREHOUSE ORDER PICKING POLICY USING GENETIC ALGORITHM*. May 26, 2020. URL: https://github.com/giacomelli/GeneticSharp/blob/master/docs/mentioning-GeneticSharp/Design-of-a-warehouse-order-picking-policy-using-genetic-algorithm.pdf.

[11] MARCUS ORDING. *Context-Sensitive Code Completion*. May 26, 2020. URL: https://www.diva-portal.org/smash/get/diva2:1088591/FULLTEXT01.pdf.

A Code and Links

The list below contains links to the code created during the project, as well as other relevant links:

- **Project GitHub Repo** - `https://github.com/Vedsted/Functional-Project`
- **GeneticSharp Github Repo** - `https://github.com/giacomelli/GeneticSharp`
- **Bug issue** - `https://github.com/giacomelli/GeneticSharp/issues/72`
- **FsCheck** - `https://fscheck.github.io/FsCheck/`
- **FsCheck GitHub Repo** - `https://github.com/fscheck/FsCheck`

B GAResult Object

```
type GAResult (GA: GeneticAlgorithm, Termination: int, Seed: int, HookChromosomes: list<IChromosome>) =
    member this.GA = GA
    member this.Termination = Termination
    member this.Seed = Seed
    member this.HookChromosomes = HookChromosomes
    member this.BestFitness = (this.GA.BestChromosome.Fitness.GetValueOrDefault())
    member this.BestGenes = (this.GA.BestChromosome.GetGenes())
    member this.printInitial = sprintf "{ Seed: %i, termination: %i}" this.Seed this.Termination
    member this.printResult = sprintf "{ termination: %i}" this.GA.GenerationsNumber
    override this.ToString () = sprintf "{ HashCode: %i, initial: %s, result: %s }" (this.GetHashCode()) this.printInitial this.printResult
```

Listing 8: GAResult Type

C Introducing Bugs Output

```
################################### Violate P1 Output ####################################
...
Label of failing property: 775 = 776
...
################################### Violate P2 Output ####################################
...
Label of failing property: Fitness decreased at index: 1
For generations 1-2: [00000000000000000000000000001000; 10000000000000000000000000000000]
...
################################### Violate P3 Output ####################################
...
Label of failing property: 1471025313.0 = 152344830.0
...
################################### Violate P4 Output ####################################
...
Label of failing property:
    [|False; False; True; True; True; False; False; True; False; True; True; False;
    True; True; True; False; True; False; False; True; True; False; True; False;
    False; False; False; False; False; True; True; False|]
    =
    [|True; True; True; False; False; False; True; True; True; False; False; False;
    True; True; False; False; True; False; True; False; False; False; False; True;
    True; False; False; True; True; False; True; True|]
...
```

Listing 9: Output when violating properties P1, P2, P3, P4

D Traveling Salesman Problem Test Output

```
1  ######################## TSP TEMPLATE ########################
2  --- Checking GAProperties ---
3  GAProperties.P1 - Number of generations match the termination criteria-Ok, passed 10000
       tests.
4  GAProperties.P2 - N+1 generation has same or better fitness than N-Falsifiable, after 55
       tests (0 shrinks) (17046819366867824694,18181908473767559667)
5  Last step was invoked with size of 6 and seed of
       (1315941839617372742,3368336864237551073):
6  Label of failing property: Fitness decreased at index: 10
7  For generations 1-11:
8   [(Distance: 8022,095889281189 Fitness: 0,5988952055359406);
9   (Distance: 8022,095889281189 Fitness: 0,5988952055359406);
10  (Distance: 8022,095889281189 Fitness: 0,5988952055359406);
11  (Distance: 7700,0392082895305 Fitness: 0,6149980395855235);
12  (Distance: 7700,0392082895305 Fitness: 0,6149980395855235);
13  (Distance: 7317,447295766522 Fitness: 0,6341276352116739);
14  (Distance: 7317,447295766522 Fitness: 0,6341276352116739);
15  (Distance: 7317,447295766522 Fitness: 0,6341276352116739);
16  (Distance: 7317,447295766522 Fitness: 0,6341276352116739);
17  (Distance: 6794,599003316314 Fitness: 0,6602700498341842);
18  (Distance: 7205,627835815638 Fitness: 0,6397186082092181)]
19 Original:
20 { HashCode: 58685418, initial: { Seed: 1868693045, termination: 57}, result: {
       termination: 57} }
21 with exception:
22 System.Exception: Expected true, got false.
23 GAProperties.P3 - Two GA's with the same inputs should result in two solutions with the
       same fitness-Ok, passed 10000 tests.
24 GAProperties.P4 - Two GA's with the same inputs should result in two identical best genes
       -Ok, passed 10000 tests.
25 occurs only with the use of a magic seed triggers an error then this will be very
       unlikely to trigger
```

Listing 10: FsCheck output

E Termination Shrinker

```
type IMPGenerators =
  static member GeneticAlgorithm() =
      {new Arbitrary<GAResult>() with
          override x.Generator = GAGen
          override x.Shrinker t =
              match t.Termination with
              | i when i < 10 -> seq{(CreateGA t.Seed (i-1))}
              | i when i < 20 -> seq{(CreateGA t.Seed (i-2))}
              | i when i < 40 -> seq{(CreateGA t.Seed (i-5))}
              | i when i < 100 -> seq{(CreateGA t.Seed (i-10))}
              | i  -> seq{(CreateGA t.Seed (i/2))}
               }
```

Listing 11: Shrinker implementation