Genetic Algorithms Revisited—A Simplification and a Free Tool for Excel Users

By Dave Snell

One of the cool aspects of teaching is that over time I start to better understand the subject I am teaching. In order to clarify a technical concept for someone else, I often find myself background processing for days or months and suddenly seeing the more obvious points—the ones often obscured by the technical details when I am learning the topic.

Based on a few years of feedback now from giving presentations to many groups, writing articles in various publications and coding programs for diverse projects, I have come to the conclusion that a genetic algorithm is a very simple concept shrouded in too many intimidating biology terms. In this article I want to share my simplified view of what a genetic algorithm is, where you might use it, how you can build your own, and how you can use the one I built for you to solve your own problems.

Basically, a genetic algorithm is a set of simple rules to solve certain types of otherwise difficult problems. In order to apply a genetic algorithm, you need a problem that lends itself to this type of solution.

Criteria that make a problem suitable for a genetic algorithm:

1. The problem involves a lot of variables—to some extent, the more variables there are, the better this technique applies.
2. Each variable can take on potential values to produce different solutions.
3. We can substitute a value for each of the variables, and that particular combination of individual values can be thought of as a solution set.
4. The problem can be quantified in some manner so that any two solution sets can easily be compared to see which is better.

In the language of our children: OMG. Can it be that simple? What about DNA, genes, chromosomes, alleles, phenotypes, mitosis and meiosis, single nucleotide polymorphisms, etc.? Didn't this all start as an attempt to mimic the amazing role that genetics plays in natural selection (unfortunately dubbed evolution)? Yes, it did.

Humans fit this set of criteria quite well:

1. The blueprint for our cells is a long chain of pairs—over 3 billion pairs in a chain.
2. Each pair can be one of four values: A-T, T-A, C-G or G-C. In order to keep this simple, I am not even going to say what the letters mean. It does not matter for this discussion.
3. Every cell contains a chain made according to these pairs. Each person has a slightly different set of links (pairs) in their personal chain.
4. The unique combinations for two different chains result in two different persons; and you can compare them to see which one is taller, thinner, smarter or whatever, to infer which chain best met your goals.

That's the end of the biology lesson!

Let's consider some applications that you might find more relevant to actuarial work:

1. You have to choose which provider groups to include in a health insurance network. There are over 3,000 provider groups in your region and each group may offer from one to 100 specialty services. Each specialty (acupuncture, cardiology, oncology, etc.) has a relative cost, and each provider group has a relative cost (specialties, location, experience, etc.). Your challenge is to pick the combination of provider groups that minimizes cost while maintaining adequate coverage for each area

CONTINUED ON **PAGE 38**

of specialty (at least five otorhinolaryngologists, at least 50 pediatricians, etc.). Since each provider group will be "in" or "out" of the network, each solution set is 3,000 values (either 0 or 1) long and your potential number of solution sets is 2^{3000}—a very large number.

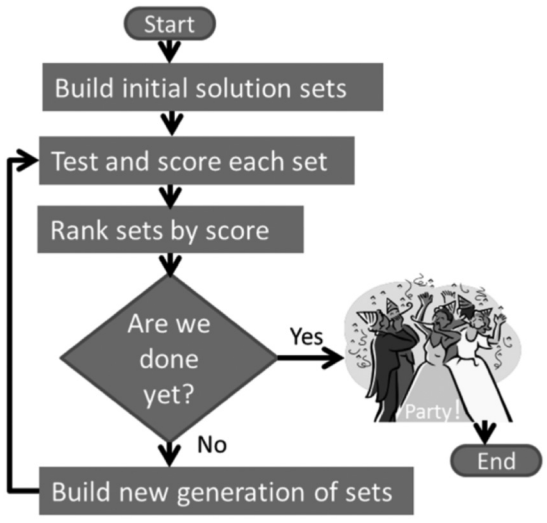
2. You have seven sales regions and 15 sales representatives. You wish to allocate them in a manner that adequately covers each region while respecting, to the extent practicable, the preference of each salesperson. Each solution set is 15 variables long, and each variable can be any of seven values so the number of solution sets possible is 7^{15} (more than 4.7 trillion).

3. You have a set of 50 equations with 100 unknowns, and the equations are not linear. Each unknown variable is a real number from the range -15 to +500. The number of potential solution sets is infinite. You want to minimize the sum of the output from each equation.

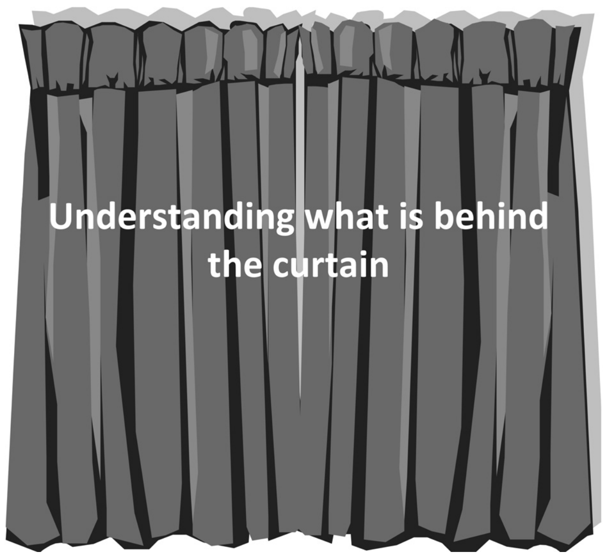
4. Your CEO asks you to do an enterprise risk management (ERM) analysis of a portfolio of business. She knows that a standard CTE (conditional tail expectation) with 10,000 stochastic runs will not surface the combination of enough tail events occurring together, yet she also knows that in the real world a tail event on one variable may trigger a domino effect where other tails are hit. Your task is to do a true worst-case type of scenario within the ranges of the several dozen key parameters.

How do we approach these problems? They all seem different; yet, they all seem to satisfy my stated criteria of applicability for a genetic algorithm.

First, I'll show you the answer:



If you do wish to learn what happens behind the curtain, please continue. Let's break our solution logic into some simple steps:



1. **Populate a collection of possible solution sets.** In previous talks and articles I have used the popular genetic algorithm term for this kind of collection as a generation. Whatever you wish to call it, the first collection of solution sets must be populated. This is one part of the process that used to prompt questions from my students. Unless you have special knowledge about the answers, it is customary to assign values to the variables of this first generation on a random basis. Most programming languages and spreadsheets offer a Random function.[1] Apply this appropriately to each variable position in your solution sets. How many solution sets should you use? Good question! If your solution set chains are very long, you may be able to fit only a smaller number of sets (perhaps 100) in memory at one time. If your chains are much shorter, you might wish to test 1,000 or more in one collection (generation). In the workbook code you can see how to do this in `PopulateInitialGeneration`.

2. **Test each set of the collection and save the scores obtained.** Whatever the nature of the problem, you need to decide how to judge the worth of your answer for a given solution set. That might be as simple as arranging your equations to end up with a single answer. Decide whether you want this answer to be minimized (for example, a cost) or maximized (a benefit). The workbook code for this is `TestSets`.

3. **Rank the scores.** Reorder the solution sets of the collection (generation) from best to worst. You can see this code in `RankTheScores`.

4. **Build the successive collection (generation) of solution sets.** This is the step that confuses the most people. If you just populate the next generation with random values, how is the method any different from a trial-and-error approach? The answer is that it is not, so we don't do it strictly randomly. Instead, we take advantage of the information from our previous collection of solution sets and results. You don't want the new collection of solution sets to ever be worse than the previous one. A way to guarantee that is to bring the top scoring sets over intact to the next collection. If you had 100 sets in your collection, you could bring over as few as one (the top scoring one) or as many as 100. There is not much point in bringing over 100 since that would really limit the improvement potential for the new collection.

If we bring over 20 solution sets from the previous generation, that means we need to create 80 more solution sets to get back up to 100 for this new generation. This is where we mimic, to some extent, biology; but again, do not let the terms of biology confuse you. We are going to build the new collection (generation) of solution sets by choosing values from the previous generation—notably the best scoring members of that generation. We might choose to pick from only the top five (those five sets with the best scores), the top 20, or even the top 50 as potential "parents" of the new generation. Let's say that each solution set chain is going to be 100 variables long. The source for variable #1 (the front of the chain) could be the variable #1 value from any solution set in our chosen group of parents from the previous generation. Often, it is most efficient to just randomly choose one of them. Likewise, the source for variable #2 in the new solution set could be variable #2 from any solution set in the parent pool. Again, just pick one at random.

"Wait a minute! Are you suggesting that a single solution set could be made from several different parent sources? Among humans, that is not allowable." You are correct! Among humans, it is not. But we were using biology only as a metaphor, so why limit ourselves to two parents per child (or two source solution sets per new solution set)?

Let's see how we could code that in Visual Basic (similar idea in most other languages):

CONTINUED ON **PAGE 40**

GENETIC ALGORITHMS REVISITED … | FROM PAGE 39

```
  Private Sub AddTheChildren()
  Dim parent As Integer, var As Long, child As Integer, children As Integer
1   children = setsPerGeneration - elites
2     For child = 1 To children
3       For var = 1 To setLength
4         parent = Int(parentPool * Rnd()) + 1
5         solutionSets(var, elites + child) = solutionSets(var, parent)
6       Next var
7     Next child
  End Sub 'AddTheChildren
```

	Detailed explanations for specific code lines
	`solutionSets(x,y)` = the value of the xth position variable in the chain for solution set y
1	For our example, we want 100 solution sets per collection (`setsPerGeneration`). We have decided to carry over intact the 20 top scoring sets (`elites`) so we need to generate 80 child (next generation aka next collection) solution sets (`children`).
2	`child` is the specific solution set you are creating
3	`var` is the variable number in the solution set of variables. `setLength` is the total number of variables in a solution set.
4	`Parent` is the specific source solution set from the previous generation. `Rnd` is a random number generator that returns a number from 0 to 1. `parentPool` is your choice for the number of parent sets. If you set `parentPool` to 50 then any of the top-scoring 50 sets could be a potential source for a `child` solution set.
5	Copy the value for variable number `var` from solution set `parent` to the new solution set `elites + child`. Again, from our example, if `child` = 1 and `parent` = 5, then copy the value from `var` 17 of old set 5 to `var` 17 of new set 21 (20+1).
6	Continue until all variables in new solution set `child` have been assigned a value.
7	Continue until all the new solution sets have been created and populated.

This gives you a new generation of solution sets. However, we can still improve our results further by randomly changing some of the child solution set values. This is called mutation. My subroutine `AddMutations` shows how to accomplish this.

Once you have a new collection (generation) of solution sets, go back and repeat steps 2, 3 and 4. Eventually, either your top-scoring solution set of step 3 is going to satisfy your goal; or it will start repeating itself. If it repeats itself for too many cycles, then it probably won't get any better so you might as well stop. If that happens before you are satisfied with the answer, then change some of the parameters: the number of solution sets in each generation, the number of elites (immortal sets), the number of parents allowed as sources, the number of generations requested, or the number of mutations allowed per set. Then, rerun the tool, choosing "from previous run" so you can continue to improve. The code `TestForCompletion` checks for a suitable end condition.

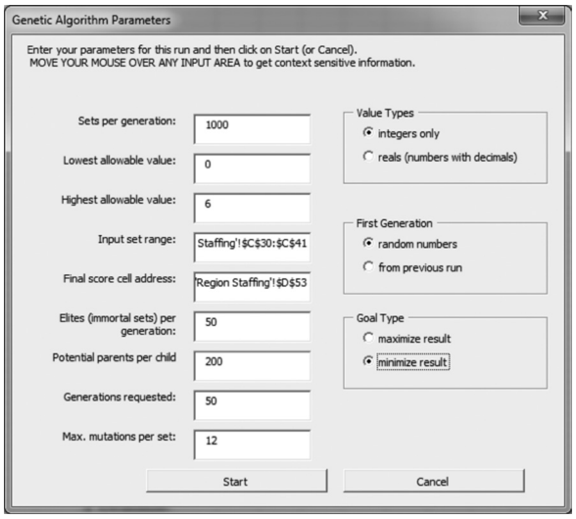
FREE TOOL

Sometimes, you don't really care how the car engine works; but you need to be able to drive it. Here is the quick way to accomplish that.

Start by downloading my general purpose genetic algorithm tool for Excel workbook from *http://www.soa.org/news-and-publications/newsletters/forecasting-futurism/default.aspx*. Save it to some folder on your PC, bring it up, and enable macros.

I won't go into a lot of detail here because the workbook has instructions built into it. The workbook also contains a couple of sample problems that you can solve to get a quick feel for how to structure your own workbook. This workbook and this article are a response to requests about how a person might adapt my earlier code to their workbook problems, I wrote this generalized genetic algorithm routine for you to be able to use it without having to learn to program. Alternatively, you can easily modify the program to extend the built-in features.

All you have to do is arrange your spreadsheets in any way that works from a given solution set (arranged in a column) and that assigns a score in some cell. On the parameters screen of the tool (see figure below), you will fill these into the "Input set range" and the "Final score cell address" (note that if you move your mouse over any input item, the program will show you context-sensitive help for that item). Then, fill in your choices for how many generations to run, how many sets per generation, how many mutations are allowed per new set, etc., and you can run your own genetic algorithm solutions.


Genetic Algorithm Parameters

Enter your parameters for this run and then click on Start (or Cancel).
MOVE YOUR MOUSE OVER ANY INPUT AREA to get context sensitive information.

Sets per generation:	1000
Lowest allowable value:	0
Highest allowable value:	6
Input set range:	Staffing'!$C$30:$C$41
Final score cell address:	'Region Staffing'!$D$53
Elites (immortal sets) per generation:	50
Potential parents per child	200
Generations requested:	50
Max. mutations per set:	12

Value Types: [✓ integers only] reals (numbers with decimals)

First Generation: [✓ random numbers] from previous run

Goal Type: maximize result [✓ minimize result]

Genetic algorithms provide you with a powerful tool for many types of problems that are very difficult to solve by other means. Recently, I discovered that Microsoft has added an "evolutionary" solution method to its excellent Excel Solver add-in. I almost stopped coding my add-in when I saw that; but then discovered that it is limited to at most 200 variables in the solution set, and it's still a black box that you cannot modify. Now that you know how to fish evolve (or, switching metaphors, now that you know what goes on behind the curtain), you are not limited to what they or I have built for you. Enjoy the free tool; and then enjoy the power of your new skill set! ▼

ENDNOTES

[1] I describe this example (from Brian Grossmiller) in detail in my article "Genetic Algorithms—Useful, Fun and Easy" in the December 2012 issue of *Forecasting & Futurism Newsletter.*

[2] Brian Grossmiller and I discussed this problem in our workshop at the 2013 SOA Annual Meeting.

[3] I am purposely assuming here that the random functions are good ones. In most applications, that is not the case; but a discussion of how you generate randomness is beyond the scope of this article. By applying it appropriately, I mean to restrict your outcomes to the range of values (either real, or integers) acceptable for that variable.

Dave Snell

Dave Snell, ASA, MAAA, is technology evangelist at RGA Reinsurance Company in Chesterfield, Mo. He can be reached at *dsnell@rgare.com*.