A Framework for Genetic Algorithms in Games

Vinícius Godoy de Mendonça Cesar Tadeu Pozzer Roberto Tadeu Raittz

[1] Universidade Positivo, Departamento de Informática
[2] Universidade Federal de Santa Maria, Departamento de Eletrônica e Computação
[3]Universidade Federal do Paraná, Curso de Tecnologia e Sistemas de Informação

Abstract

This article describes the architecture of a Genetic Algorithm Framework, techniques and optimizations used in its construction, a new selection algorithm called Asymptotic Selection, considerations about the use of this process in games and a usage example.

Keywords: artificial intelligence, genetic algorithms, biological programming, optimization techniques, evolutionary computation

Authors' contact:
vinigodoy@gmail.com
pozzer@inf.ufsm.br
raittz@ufpr.br

1. Introduction

Artificial Intelligence (AI) for Games often needs to deal with environments composed by several variables with unknown exact behavior. Consider a game like Sim City or Civilization, where the game AI advisor could suggest for the player an administrative strategy. This strategy should try to maximize several factors like the incoming money, production rate and technological advancements while reducing problems like pollution and famine. The consequence of each variable to the global scenario is not entirely predictable but, usually, given a set of defined values, it's easy to evaluate the whole picture.

Traditional AI offers several search mechanisms [Russel and Norvig 2003] which tries to find these answers. Some of them tries to systematically search the entire search space (like breadth-first search or A*). While they surely lead to the best possible solution, they may have a very long execution time. Another option would be using a local space search, like hill climbing search or Tabu search, but they tend to be locked in local maximums.

Genetic Algorithms (GA), are in the last category of the above two and try to use the principle of evolution, normally found in natural systems, to search for solutions to algorithmic problems [Schwab 2004].

The classical genetic algorithm defined by Holland [Holland 1975] is divided into six steps [Charles et al. 2008]: population creation, fitness evaluation, chromosome mating and mutation, deletion of lowest fitness individuals and re-evaluation of the new population. The last steps are repeated until the convergence of the population to a good result.

One of the greatest advantages of GA is that it can be considered a parallel search mechanism that tests several different variables simultaneously [Charles et al. 2008]. Also, it avoids local maximums by doing mutations, and allowing searching of random areas of the solution space.

Unfortunately, this technique is not well known among the game developer community. There are few books on the subject and even less containing practical game situation examples. This article presents a framework that allows developers to easily integrate genetic algorithms into their games. The proposed solution is both flexible and easy to use.

The article also discusses some optimizations made in the framework. To improve understanding of the proposed architecture, a sample test scenario is provided in section 4.

2. Drawbacks of Genetic Algorithms

Time consuming evolution: Often evolution takes too many generations, even with a good genome designed with good operators [Schwab, 2004]. This is especially true in the presence of *deceptive problems* [Charles et al. 2008].

Evaluation by experimentation: Due to the great number of crossover, mutation, selection and scaling options, and due to the random nature of GAs, the only way to find a good solution is by experimentation [Schwab, 2004]. This also implies that a good genetic algorithm framework must ensure that experimenting is possible.

No guarantee of optimal solution: Just like any stochastic selection algorithm [Russel and Norvig 2003], there's no guarantee that the algorithm truly converged to the global maximum. On the other hand, for games finding good local maximum could be as good as choosing the global maximum especially because it could create a more human behavior [Schwab, 2004].

Hard to debug: Due to the random nature of the algorithm, it's hard to tell why a given implementation is not working.

3. Framework description

This section describes the entire framework organization and design considerations.

3.1. Domain specific classes

The first step when dealing with any genetic algorithm is to represent the problem in a form of a genome structure, and provide a fitness function to evaluate a specific individual.

In the proposed framework, there's a clear distinction between the concepts of an *Individual* and of a *Genome*. Such is not the case in several implementations like Schwab [2004], Charles et al. [2008] Jones [2008] and Obitko[1998]. So, the *Individual* has three very important roles:

1. To provide business methods for accessing domain specific properties;
2. To implement a common GA interface, allowing the framework to work;
3. To act as a *Mediator* [Gamma et al., 1995], for its genome structure and manipulation.

Notice that the individual models the business problem, so, it should be implemented by the framework user.

This approach has two major advantages:

1. The fitness function may use user-friendly methods to evaluate an individual, not manipulating the genome directly. This also makes the fitness function fully tolerant to genome structure changes;
2. The genetic framework does not depend upon a specific genome structure. User defined structures may be created and different implementations can be tested;

The fitness function is specified by an interface, and in case of C++ implementations a *functor* [Vandevoorde and Josuttis 2003] may be used instead.

3.2. Genome classes

The framework already provides the BitGenome class, that helps user's to model the problem as a bit string. This is the most common method [Schwab 2004 and Jones 2008]. BitGenome makes use of the *Strategy* design pattern [Gamma et al. 1995] for its crossover and mutation methods.

Two implementations of crossover are already provided: the point crossover method and a uniform crossover. Point crossover method provides a user-defined number of crossover points, and could be used as a single or multi point crossover. For the mutation method, only the random mutation is provided.

The class diagram below describes this structure:

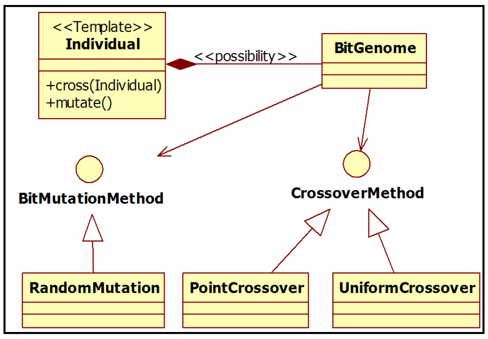

Figure 2: Bit genome –a possible individual internal structure

3.3. Scaling classes

Scaling is a common and optional technique to avoid the so called "super individuals" [Schwab 2004, Charles et al. 2008], that is, individuals with a score so high that they easily dominate the entire population. These may be a problem, since they may represent just a local maximum. The framework provides two common kinds of scaling: a rank scale and a sigma truncation scale [Schwab, 2004].

3.4. Selection function

The framework provides three selection methods proposed by Schwab [2004]: Tournament Selection, Roulette Wheel Selection and Stochastic Selection.

The traditional roulette wheel selection is described as follows [Obitko, 1998]:

```
S ← sum_of_all_fitness_scores(population)
r ← random(0, S)
for each individual in population
    s ← s +individual.fitness()
    if (s > r) then return individual
next
```

Figure 3: Traditional Roulette Wheel Selection

Exactly the same implementation is also found in Jones, 2008. This implementation assumes that the population is sorted in descending order.

The problem with this approach is that iterates through the population every time an individual must be selected. Since iteration is a time-consuming task, using this approach could be time killing for games. So an optimized implementation of the same algorithm was provided, allowing the iteration to occur only once:

```
S ← sum_of_all_fitness_scores(population)
for int i = 1 to desired_individuals_count do
    r[i] ← random(0,S)
next i
sort(r) //ascending order
lowest ← 0, sum ← 0, i ← 0
for individual in population
    s ← s + individual.fitness()
    while (r[lowest] < s)
        selected[i] ← individual
        i ← i+1
        lowest ← lowest+1
    wend
next
return selected
```

Figure 4: Optimized roulette wheel selection

3. The generation class also provides several defaults, based in the common options [Schwab, 2004]. These are: a crossover rate of 75%, an elitism rate of 5%, a, mutation rate of 100% (but it considers that user will use the BitGenome, which default is 1% per bit), Roulette Wheel Selection and no scaling;

4. BitGenome fully implements the Individual interface, so it could be used directly. Off course, this will remove the benefits described in section 3.1, but allow a very simple usage of the framework;

4. Usage sample

To exemplify the usage of the framework, let's consider a Sim City like game where the player must manage the plantation of sugar maple (for biodiesel) and food. Player uses industry and food taxes to stimulate or decelerate each one of these plantation options.

Increasing the sugar maple fields will stimulate the industry, so the player will earn more money with taxes and the population will have more jobs. On the other hand, too many maple fields mean that the food will get more expensive, and it's quite possible that some people will get hungry. Increasing the food fields allow the player to make food cheaper, but it will slow down the industry. This could generate unemployment, and no working people will not buy any food or pay any taxes. They also generate other problems, such as crimes, not considered in our analysis. The exact impact of each parameter variation was configured by the game designer, in a script.

Now, let's consider an implementation of an advisor that suggests for the player a good approach about how to configure these taxes.

The first step is to configure an individual that contains all parameters that can be changed. They are the biodiesel industry tax and the food tax. We use the BitGenome structure to represent these taxes. The first 7 bits are used to store the industry tax and the last 7 bits to store the food tax.

After that, we will create the Fitness Function. The fitness function will use a game function that given the two taxes calculate the incoming money, rate of hungry people and rate of unemployed people. Notice that this function is commonly found in this kind of game, since it is also used for the game mechanics and to give support to the player decision.

The advisor will prefer individuals that contain:

1. No unemployment: People that don't work are hungry, do not contribute to the incoming taxes, and generate social problems, so they will be avoided to the maximum;
2. The biggest income as possible;
3. Little famine;

So, our fitness function could be:

```
unsigned calculate(Individual individual) {
   //First, discard invalid taxes
   if (individual.getIndustryTax() > 100)
       return 0;
   if (individual.getFoodTax() > 100)
      return 0;

   //Then calculate the projection
   Projection proj = calculateProjection(
          individual.getIndustryTax(),
          individual.getFoodTax());

   //Finally, calculate the score
   int score = proj.earnedMoney *
          1 - (proj.famineRate() +
          5 * proj.unemploymentRate());

   //Limit the lowest score to zero.
   return (unsigned) max(0, score);
}
```

Figure 5: Advisor fitness function

We can see that this advisor rejects unemployment five times more than famine. Also, earned money has a linear impact over the overall score and this impact is amortized by famine and unemployment.

Finally, let's include a sample of the "suggest" function. The function will use the genetic algorithm defaults. Taxes is our individual class. We will use an optional generation constructor that receives an *Abstract Factory* [Gamma et al. 1995] and generate n individuals using this factory.

```
Taxes Advisor::suggest() {
   //Create a generation
   Generation<Taxes> generation(factory, 30);

   //Calls next() many times, for 500ms
   generation.nextUntilTime(500);

   //Returns the best individual
   return generation.bestIndividual();
}
```

Figure 6: A sample advisor implementation

Even if a local maximum is returned, it has good chances to be a good advice. That's ok for this implementation, since a human advisor is not perfect either, and could not see the best solution sometimes.

5. Conclusion

In this work, a genetic algorithm framework was presented, as well as its possible uses for games. Some optimization techniques were described, as well as several design considerations over the framework.

The asymptotic selection algorithm was presented, which gives advantages over speed when avoid scaling while using a simple implementation and easy of use.

The framework also presented a very flexible and customizable architecture, ideal for games. Most part of related works are suited for other areas like Galapagos (education) [AndrewMan, 2008] or Heat Exchangers [Tayal et. al 1998]. Similar work can be found for Java in JGAP [Meffert, 2008], but it does not leave a clear distinction in individual roles, like proposed here.

There are several options for future work, including increasing algorithm options in every step, introducing parallel processing and thread-safety to the framework or even creating self-adapting genome schemes.

The framework implementation is public and can be found in: http://sofiaia.sourceforge.net

References

ANDREWMEN, GalapagosGA Framework. Available from: http://sourceforge.net/projects/galapagosga [accessed August, 2008]

CHARLES, D., FYFE, C., LIVINGSTONE D., MCGLINCHEY, S., Biologically Inspired Artificial Intelligence for Computer Games, *IGI Publishing*, 105-138.

JONES, M. T., 2008. Artificial Intelligence: A Systems Approach. *Infinity Science Press Inc.*, 195-247.

SMED, J., HAKONNEN, H., 2006. Algorithms and Networking for Computer Games. *John Willey and Sons*, 205.

SCHWAB, B., 2004, AI Game Engine Programming, *Thomson Delmar Learning*, 413-452.

NORVIG, P. AND RUSSEL, S.J., 2003. Artificial Intelligence: A Modern Approach, *Prentice Hall*, 116-119.

VANDEVOORDE D., JOSUTTIS N. M., 2003. C++ Templates – The complete guide, *Addison Wesley*, 417-474

OBITKO., MAREK. 1998. Introduction to Genetic Algorithms, [online],Hochschule für Technik und Wirtschaft Dresden. Available from: http://obitko.com/tutorials/genetic-algorithms/ [accessed August, 2008].

GAMMA E., RICHARD H. JOHNSON, R, VLISSIDES, J. 1995. Design Patterns – Elements of reusable object-oriented software, *Addinson-Wesley Longman Inc.*

HOLLAND, J. H., 1975. Adaptation in natural and artificial systems, *Ann Arbor MI*, *University of Michigan Press*.

TAYAL, MANYSH C., FU YAN, DIWEKAR URMILLA, Optimal for Heat Exchangers: A genetic algorithm framework. http://pubs.acs.org/cgi-bin/abstract.cgi/iecred/1999/38/i02/abs/ie980308n.html [accessed August, 2008].

MEFFERT, K., Java Generic Algorithm Platform (JGAP), Available from: http://jgap.sourceforge.net/ [accessed August, 2008].