Designing a Framework for Genetic Algorithms by Contract in Python
CSI 5110 Semester Project

Ashwin Panchapakesan
apanc006@uottawa.ca

Dec 05, 2012

1 Motivation

Genetic algorithms (GAs) are a class of algorithms that try to mimic natural, biological evolution *a la* Darwinian natural selection, to compute solutions to a given problem. They are especially useful when no well known strategies for computing solutions to such a problem exist. Though these algorithms have been studied since the 1950s [1], it wasn't until John Holland's work in the early 1970s that they gained popularity [2].

Genetic algorithms begin by creating a collection (population) of random potential solutions (chromosomes). Over time, chromosomes in the population are selected in a manner proportional to their "fitness"[1] for crossover and mutation operators. These operators have the effect of combing the best solutions with each other in order to make better solutions. It is through the application of these operators that the population evolves into the next generation.

One important problem facing code implementing GAs is that due to the dynamic nature of the individual chromosomes in a population, the introduction of bugs into source code becomes too easy. This problem is only exacerbated when attempting to code GAs in a dynamically typed language. This problem is further aggravated when coding GAs in Python, whose design philosophy it is, that as many expressions in code as possible should compute to some meaningful result or other [4].

As a result, most of the errors that a programmer sees when building or using a genetic framework in Python are far too cryptic[2], even given Python's excellent error reporting. The reason for this is that any error that Python reports is due to an expression that it is unable to compute. This does not give

[1] a measure calculated by an objective function, reflecting how well the chromosome would solve the problem at hand

[2] The error message `AttributeError: 'list' object has no attribute 'chromosomes'` is perfectly reasonable and well written. However, in the context of a GA, it is very unclear which part of the code is ausing this error.

the programmer much insight as to which part of the framework caused such an error. Debugging such an error requires spending time analyzing different components of the population as they undergo genetic operations and examining the output - a tedious task to say the least. Further, since GAs rely heavily on chance[3], such a problem might not be reproducible very easily.

Thus, a more thorough debugging tool is required. This problem lends itself very well to be solved by implementing Design by Contract (DbC). Applying an existing design-by-contract model checker (called PyContract [8]) to an existing, open-source GA framework [7] in order to improve the usability of the framework. The framework allows programmers to write their own functions for genetic operators, population initialization, etc. Applying design-by-contract to this framework would ensure that problems in the design of a GA (to be implemented using this framework) such as improper mutation or crossover operators would be caught before the evolution runs. Thus, any unfavorable outcome of the result of evolution would be due to functions not doing handling inputs or returning outputs that diverge from the expectations of the framework.

2 Introducing Genetic Algorithms

To explain the functional principles of genetic algorithms (GAs) in this section, I will use the example of the Traveling Salesman Problem on the well known Berlin-52 map, which contains 52 cities.

A GA begins by generating an initial population of random tours for the traveling salesman. Once this initial population is created, it selects individuals to probabilistically crossover and mutate, thereby making child individuals, which comprise the next generation of this population. Repeating this process of selection, crossover and mutation over several generations allows the GA to converge on an optimal solution.

2.1 An Individual

Individuals in a GA to solve this problem are made up of one chromosome. This chromosome is an `list` of 52 integers, each one representing a city on the map. In order for an individual to represent a legal solution in the solution space, the chromosome is a permutation of $\{0, 1, 2, ..., 51\}$, thus making it a valid tour for the traveling salesman problem.

2.2 Fitness of an Individual

Since the optimal solution for this problem is an individual whose tour length is minimal, the fitness of each individual could be the length of the tour it represents. However, since the goal is to maximize the fitness, a better fitness

[3] classically, crossover occurs with high probability and mutation occurs with low probability

measure of an individual would be the negative of the length of the tour it represents. This can be easily computed under the following assumptions:

1. Each of the 52 cities on the map is represented as a point on the xy plane
2. There is a straight line (road) connecting every pair of the 52 cities on the map

Thus, the fitness of an individual can be computed as follows:
$-dist(city_{51}, city_0) - \sum_{c=0}^{50} dist(city_c, city_{c+1})$
where
$dist(c_i, c_j) = \sqrt{(city_i.x - city_j.x)^2 + (city_i.y - city_j.y)^2}$

2.3 Selecting Individuals

In order to create new individuals out of existing individuals, two are selected for crossover and mutation operations. The selection mechanism is fitness proportional, meaning that individuals with higher fitness are selected more often than individuals with lower fitness.

2.4 Crossover

A crossover is an operation that takes two parent individuals and creates a new child individual, whose chromosomes are comprised of parts of the corresponding chromosomes from both parents. In the case of this traveling salesman problem, one possible crossover function is defined as follows:

1. Select points A and B such that $0 < A < B < 51$
2. Make an empty child chromosome which is intended to hold 52 cities (a new tour for the traveling salesman)
3. Copy over all the cities between points A and B in the tour represented by $parent_1$ into the child chromosome
4. Copy over all the cities before point A and after point B in the tour represented by $parent_2$ into the corresponding location in the child chromosome, as long as the city does not already exist between points A and B in the child chromosome.
5. Fill in the remaining cities in the child chromosome based on the order in which they appear in $parent_1$.
6. Insert this child chromosome into a new individual - the child individual of the crossover.

Note that it is imperative that the child individual of a crossover represent a legal solution so as to ensure that the GA does not create individuals that are outside the solution space.

2.5 Mutation

A mutation is an operation that slightly changes an individual. One possible mutation is to swap the positions of two cities in the tour. Another possible mutation is to reverse the order of the cities in one contiguous part of the tour.

Note that it is imperative that a mutated individual must still represent a legal solution so as to ensure that the GA does not create individuals that are outside the solution space.

3 Introducing Design by Contract

3.1 The Design by Contract Principle

Design by contract (DbC) is the principle that interfaces between modules of a software system should be governed by precise specifications. The contracts will cover mutual obligations (preconditions), benefits (postconditions), and consistency constraints (invariants) [3]. This principle is especially applicable in large modular systems with multiple levels of abstraction, such as a framework for implementing GAs.

3.2 The Advantage of Using DbC in this Framework

Two of the core design principles of the python programming language are

1. Almost all expressions that a programmer tries to compute must be computed in some meaningful way.

2. All error reporting and tracebacks should be meaningful in order to help a programmer better debug their program. In particular, core-dumps and crashes should be avoided.

In most cases, these are very desirable principles in a programming language. However, when working with GAs, where even simple off-by-one errors can cause individual solutions in a population to leave the solution space and where mutation and crossover operations are probabilistic, bugs become difficult to reproduce and traditional step-through debugging becomes infeasible (except in a small subset of the program's functional body). While this error reporting explains why the GA may crash, it does very little to reveal the real source of the error (for example, it may be clear that two variables of very different data types may not be added together, but the bug that causes either variable to be of that different datatype is not identified). Thus, whereas the error messages may be well written for most other algorithms, they are rendered far too cryptic to help debug a GA.

Further, due to the amount of data that a GA works with on the stack, traditional print-debugging (or logging) would also be infeasible as the signal-to-noise ratio in the debug logs would be too small to be useful to a programmer.

One particularly difficult bug was found to be caused by mistyping `if a>b: a,b = b,a` as `if a<b: a,b = b,a` in the crosover function for the GA solving the traveling salesman problem. This had the effect of causing this error, mid evolution: `IndexError: pop from empty list`. This is because the segment between points A and B (refer to section 2.4 for the explanation of the crossover function; also not that in the implementation, variables `a` and `b` refer to A and B) evaluates to an empty list in the current python implementation if `a` is set to be larger than `b`. As a result, wrong cities are copied over from the segments before A and after B and there are fewer "remainder cities" than empty slots in the tour. However, the raised `IndexError` does not provide any information as to the source of this error.

Further, assuming that the GA runs without any errors, if the end result of a run of the GA is unfavorable or unexpected, it is unclear as to whether this divergence in expectations was caused by the stochastic nature of evolution (and incorrectly programmed parameters thereof) or by faulty programming logic. As a result, implementing contracts for each of the functions in this framework allows programmers to catch errors in programming logic early, trace the error to the buggy function in the program, and eliminate programming errors as the reason for unexpected results at the end of a run of the GA (assuming that the correct contracts have been implemented).

4 DbC Frameworks for Python

In order to implement DbC to this GA framework, two packages were evaluated. These packages are reviewed in this section

4.1 PyContracts

PyContracts is a DbC package that allows a programmer to annotate functions with contract expressions. The syntax of PyContracts follows closely, standard Python ReSTful documentation [5]. This makes the documentation itself serve two purposes:

1. the documentation for the function, which can then be automatically generated into official documentation by software such as Sphinx
2. the contract expressions for DbC.

For example, the contract expressions in algorithm 1 for a function that multiplies two matrices[5] denotes that:

1. The input parameter `a` is an `array` of dimensions `M` rows and `N` columns, where `M` and `N` are positive numbers.
2. The input parameter `b` is an `array` of dimensions `N` rows and `P` columns, where `P` is a positive number.
   (a) This ensures that `a` and `b` are of compatible dimensionalities.

Algorithm 1 Contracts for a Matrix Multiplication Function in PyContracts

```
@contract
def matrix_multiply(a, b):
    ''' Multiplies two matrices together.
        :param a: The first matrix. Must be a 2D array.
          :type a: array[MxN], M>0, N>0
        :param b: The second matrix.
                  Must be of compatible dimensions.
          :type b: array[NxP], P>0

           :rtype: array[MxP]
    '''
```

3. The function returns an array of `M` rows and `P` columns.

Note however, that this set of contracts for this function does not mention anything about the values of the inputs remaining unchanged over the execution of the function; nor is there a way to check this. In fact, PyContracts does not allow programmers to write contracts about the values of variables before the function is run, to be checked after the execution of the function [6].

Upon further investigation, it is clear that PyContracts does not save variables on the stack at different points in time during execution. This implies that it is not possible to write contracts for post-conditions that express properties about the values variables before and after the function's execution. For the same reason, it is also impossible to write contracts that express properties of invariants.

As a result of these limitations, PyContracts is not a suitable DbC package for writing contracts for the GA framework.

4.2 PyContract

PyContract is a DbC package that allows a programmer to annotate functions with contract expressions. Although its syntax is unlike PyContract's, its syntax does allow for the development of richer contract expressions, making up for PyContracts' shortcomings.

For example, the contract expressions in algorithm 2 for a function that multiplies two matrices denotes that:

1. The input parameter `a` is an `array` of positive row and column dimensions.
2. The input parameter `b` is an `array` of positive row and column dimensions.
3. The number of rows in `b` is equal to the number of columns in `a`.
   (a) This ensures that `a` and `b` are of compatible dimensionalities.

Algorithm 2 Contracts for a Matrix Multiplication Function in PyContract

```
def matrix_multiply(a, b):
    ''' Multiplies two matrices together.
        pre:
            isinstance(a, array)
            isinstance(b, array)
            len(a) > 0
            len(a[0]) > 0
            len(b) == len(a[0])
        post:
            __old__.a == a
            __old__.b == b
            isinstance(__return__, array)
            len(__return__) == len(a)
            len(__return__[0]) == len(b[0])
    '''
```

4. The function returns an array of `M` rows and `P` columns.
5. The inputs are unchanged

In addition, PyContract also allows for the expression of class invariants, thus overcoming another one of PyContracts' shortcomings. These are expressible with the `inv` declaration in the contract expressions along with referencing `self`. However, there is a limitation to PyContract's ability to express invariants in functions. For example, it is not possible to express contracts about loop invariants whose expressions contain variables that are not class variables, but are instead local to the scope of the function itself. In order to express such invariants in this GA framework, a hybrid approach using both PyContract and assert statements native to Python were used.

5 DbC Implementation

As previously stated, the DbC implementation for this GA framework is a hybrid of PyContract and `assert` statements native to the python programming language. For example, the crossover function described in section 2.4 has the contracts shown in algorithm 3 (explained in table 1).

Notice that there are no contracts that express invariants in the PyContract syntax. This is because contractual invariant clauses may express invariants that refer to only class variables. This does not include variables that are not bound to a defined class but are still within the local scope of the function for which the contract is written. Therefore, loop invariants that refer to loop counters cannot be checked using PyContract. As a result, the second part of

Algorithm 3 Contracts for the Crossover function for the Traveling Salesman Problem

```
def injectionco(p1, p2):
    """
        pre:
            isinstance(p1, list)
            isinstance(p2, list)
            len(p1) == len(p2)
            sorted(p1) == range(len(p1))
            sorted(p2) == range(len(p2))

        post[p1, p2]:
            p1 == __old__.p1
            p2 == __old__.p2
        post:
            isinstance(__return__, list)
            len(__return__) == len(p1)
            id(__return__) not in [id(p1), id(p2)]
            forall(__return__, lambda city: city in p1 and city in p2)
            len(set(__return__)) == len(__return__)
    """
```

Table 1: Explanations of Contracts in Algorithm 3

Contract Expression	**Explanation**
`pre:`	The next block of indented expressions are pre-conditions of this function
`isinstance(p1, list)`	`p1` is a list
`isinstance(p2, list)`	`p2` is a list
`len(p1) == len(p2)`	`p1` and `p2` have equal number of elements
`sorted(p1) == range(len(p1))`	`p1` is a perumatation of $\{0, 1, 2, ..., L-1\}$ where `L` is the number in elements in `p1`
`sorted(p2) == range(len(p2))`	`p2` is a perumatation of $\{0, 1, 2, ..., L-1\}$ where `L` is the number in elements in `p2`
`post[p1, p2]:`	The next block of indented expressions are post-conditions of this function on the variables `p1` and `p2`
`p1 == __old__.p1`	`p1` remains unchaged as a result of executing this function
`p2 == __old__.p2`	`p2` remains unchaged as a result of executing this function
`post:`	The next block of indented expressions are general post-conditions of this function
`isinstance(__return__, list)`	A `list` is returned
`len(__return__) == len(p1)`	The number of elements in the returned list is equal to the number of elements in `p1`
`id(__return__) not in [id(p1), id(p2)]`	The returned list does not reside in the same memory location as either `p1` or `p2`
`forall(__return__, lambda city: city in p1 and city in p2)`	Every element in the returned list exists in `both p1` and `p2`
`len(set(__return__))== len(__return__)`	Every element in the returned list occurs exactly once

Algorithm 4 Contracts for the Crossover function for the Traveling Salesman Problem

```
def runTSPGA(kwargs):
    ...
    while g < maxGens:
        if testmode:
            assert g < maxGens
            assert best[1] < targetscore
    ...
```

the hybrid implementation of contract checking uses `assert` statements native to python to enforce invariants which express properties of non-class-variables within the local scope of the function. For example, contractual loop invariants are expressed in algorithm 4.

These loop invariant contract expressions check to ensure that the invariant is True and the hyp. guard is False in every iteration of the while-loop.

It is important to note that GAs themselves usually have a long runtime. Furthermore, contract checking implies that every time a function is called, the pre-conditions, post-conditions and invarians of that function are verified. In addition, checking post-conditions in this framework, especially against values of variables before the execution of the function requires making a copy of the stack before each execution. The result is a drastic increase in the runtime of these functions, explicitly because of contract checking. In order to alleviate such effects of DbC on the GA framework, a new configuration parameter was introduced into the framework. This parameter (named `testmode`) is a boolean flag, which when set `True` forces contract checking on all functions for which a contract has been written. When this flag is set `False`, the contracts are not checked, allowing the GA framework to operate at its maximal efficiency without being interrupted by contract checking [7]. Therefore, the ideal usage of a simulation using this framework (now augmented with DbC) would be to run the simulation once for a very short period of time (evolutionary time, not realtime) to ensure that all contracts are being followed. Once it is clear that all contracts are being followed, then the simulation may be run for the required (presumably longer) period of time without contract checking, so that it may run at an efficiency that is not hindered by contract checking.

A list of all contracts implemented in this framework is available in the official documentation of the Pyvolution package [7].

References

[1] Barricelli, Nils Aall. "Esempi numerici di processi di evoluzione." Methodos 6.21-22 (1954): 45-68.

[2] Holland, John H. "Adaptation in natural and artificial systems, University of Michigan press." Ann Arbor, MI 1.97 (1975): 5.

[3] Jazequel, J-M., and Bertrand Meyer. "Design by contract: The lessons of Ariane." Computer 30.1 (1997): 129-130.

[4] van Rossum, Guido. "Guido van Rossum on the History of Python." Guido van Rossum on the History of Python. Dropbox. California, San Francisco. 20 2011. Speech.

[5] Censi, Andrea. "PyContracts 1.4.0." PyContracts 1.4.0dev documentation. Python.org, 10 2012. Web. 5 Dec 2012. <http://andreacensi.GitHub.com/contracts/>.

[6] Censi, Andrea. "Need help with PyContracts." 20 2012. Reddit, Online Posting to reddit.com. Web. 5 Dec. 2012. <http://www.reddit.com/r/Python/comments/11szm3/need_help_with_pycontracts/c6ph64p>.

[7] Panchapakesan, Ashwin. "Pyvolution 1.1." Genetic 1.1 documentation. Python.org, 12 2012. Web. 5 Dec 2012. <http://packages.python.org/Pyvolution/settings.py.html#fields>.

[8] Webb, Jason. "A declarative data contract container type for Python." pycontract 0.1.4 (2010). Python Package Index. Web. 5 Dec 2012. <http://pypi.python.org/pypi/pycontract/0.1.4>.