Genetic algorithms: optional assignment

Optimization

Nieves Montes Gómez (1393150)

07/01/2020

1 Introduction

The objective of this assignment is to find the maximum value of a very ugly function f using a basic version of the genetic algorithm (GA). Such function is defined on D^4, where D is the set of non-negative integers between 0 and $2^{32} - 1$.

In order to achieve this objective, I will be implementing a basic version of the GA in C++. The complete source code can be found here.

2 Implementation

The approach to solve this problem relies on object-oriented programming. The code is organized in four files:

1. `main.cpp` contains the `main()` function as well as the instantiation of all objects related to random number generation.
2. `class_individuals.hpp` defines the class `individuals`, all its attributes, member and non-member functions.
3. `class_population.hpp` defines the class `population`, its attributes and member functions (it does not have any non-member functions).
4. `ugly_function.hpp` contains the definitions related to the problem function whose maximum we are trying to find.

2.1 Class `individual`

Candidate solutions are represented by instances of the class `individual`. They have four `unsigned long` attributes which correspond to the variables x, y, z and t, the arguments to the function we are trying to maximize. As `unsigned long` variables, they can be interpreted as the phenotype, while their binary representation is identified as the genotype or chromosomes.

The attribute `fitness` corresponds to the image of the phenotype under function f. By looking for the fittest individual, we select that whose fitness score corresponds to the suspected maximum of f.

Class `individual` has the following member functions:

- `random_individual()` sets the values of the phenotype variables to random `unsigned long` integers between 0 and $2^{32} - 1$. More information on this in section 2.3.
- `find_fitness()` computes the fitness score of an individual from its phenotype variables.
- `mutate()` introduces random mutations. It does so through the standard bit-flip mutation procedure (Algorithm 22 [1]). The probability of mutation for each of the $32 \times 4 = 128$ positions is controlled through the variables `p_mut`. In order to increase the explorability character of the algorithm, `p_mut` is set to 0.01, which is slightly higher than the standard for bit-flip mutation ($\frac{1}{128} \sim 0.007$). Finally, the fitness score is recomputed by calling `find_fitness()`.
- `print_individual()` outputs the attributes of an individual both in integer and binary representations. It is used for debugging and to output the solution at the end of execution.

The class `individual` also contains the non-member function `one_point_crossover()`. It takes in a pointer to two individuals and outputs a vector of two more, their offspring. The breeding method of choice is one-point crossover. The position `c` where the parents' chromosomes are cut and mixed is newly generated for each chromosome. The genotypes of the children are allowed to mutate by calling the `mutate()` function before computing their fitness and being returned.

2.2 Class `population`

Class `population` represents a generation of candidate solutions. Its attributes are a vector of `individual` objects called `members`, and a pointer to the fittest individual. This class contains a parametrized constructor that takes in the population size (`popsize`), generates the `members` vector randomly, looks for the fittest individual and sets a pointer to it.

Class `population` has the following member functions:

- `find_fittest()` returns a pointer to the fittest individuals in the `members` vector.
- `tournament()` implements tournament selection (Algorithm 32 [1]). This is the selection algorithm of choice as it is the most widely used in GA problems given its simplicity and insensibility to the fitness score. It randomly selects two different individuals from the `members` vector and returns a pointer to that with the highest fitness function.
- `new_generation()` replaces the current `members` vector with another, named `Q` in the scope of the function. For `popsize`/2 times, it calls the `tournament()` function twice to select two different parents, breeds them by calling `one_point_crossover()` and pushes the two new offspring into the `Q` vector. Finally, it substitutes the `members` vector with `Q` and calls `find_fittest()` to find the new fittest individual.

2.3 `main()` function

The `main()` function is defined in file `main.cpp`. It contains all necessary imports and the definitions of the constant expressions:

- `N = 32`. This value is set by the requirements of the problem.
- `popsize = 1000` corresponds to the population size. This value is considered to be a good compromise between a population too small that may lead to very low variability in the genetic pool, and a population too large that takes a very long time to update.

- `p_mut = 0.01` corresponds to the probability of mutating any given bit position in the variables x, y, z and t. The particular value of this variable has been previously commented (see this section).

- `update_limit = 200` is the maximum number of times the population is allowed to produce new generations without updating the current solution. For example, let's suppose that a very fit individual is found during iteration 50. If for the following `update_limit` iterations the fittest individuals of the new generations are not better solutions, the algorithms halts.

The `main.cpp` file also contains all instances of objects related to the generation of random numbers. It relies on the classes defined on the C++ header `random`. First, the `random_device` object `rndgen` is instantiated. It feeds from the Linux devices `/dev/urandom` or `/dev/random` to generate uniformly distributed true random numbers between 0 and $2^{32}-1$, which matches the requirements of the problem. Both `/dev/urandom` and `/dev/random` feed from the background noise of the hardware. `/dev/random` blocks when there is not enough entropy, however this check can slow considerably the program. Given this drawback, `/dev/urandom` is the random generator of choice, despite not having this entropy check.

The raw numbers produced by the `random_device` are used to initialize the x, y, z and t variables when random individuals are produced. This device also feeds `uniform_real_distribution unif()`, which outputs random double numbers between 0 and 1. This is used on a variety of occasions: to mutate individuals in `mutate()`, to set the crossover points in `one_point_crossover()` and to select the parent candidates in `tournament()`.

The body of the `main()` function is very straight-forward. After defining all necessary variables including a randomly initialized `population`, the candidates are allowed to generate new generations. The provisional solution is updated if the new generation contains a fitter individual. The loop is exited as explained by the restriction imposed by `update_limit`. Finally, the employed parameters and the ideal solution are printed.

3 Results

Following are some outputs produced by running the program with the following parameters: `popsize = 1000`, `p_mut = 0.01` and `update_limit = 200`.

Run 1:

```
fittest individual found during the 89-th iteration

genotype:
chX: 11110101001101001000010001010110
chY: 11110010100000011011100010001110
chZ: 00000000011101111100100011001000
chT: 00000010110101111011001010101000

phenotype:
x: 4113859670
y: 4068587662
z: 7850184
t: 47690408

fitness: 8.49278e+77
```

Run 2:

```
fittest individual found during the 134-th iteration

genotype:
chX: 11111011100010110010110010111010
chY: 11111100110111101000010110011001
chZ: 00000000000111110111111111111000
chT: 00001000110011000010011101001011

phenotype:
x: 4220202170
y: 4242441625
z: 2064376
t: 147597131

fitness: 9.35699e+77
```

Run 3:

```
fittest individual found during the 96-th iteration

genotype:
chX: 00010110101011010000001010101001
chY: 11111111011001010000110000111111
chZ: 11110100111111100100000000010100
chT: 00000100110010111101011111001000

phenotype:
x: 380437161
y: 4284812351
z: 4110303252
t: 80467912

fitness: 7.12401e+77
```

Run 4:

```
fittest individual found during the 89-th iteration

genotype:
chX: 11110111100011100010101101010111
chY: 11111111010100011011011100101111
chZ: 00001110001011000010011110100011
chT: 00000011111001111001001110000100

phenotype:
x: 4153289559
y: 4283545391
z: 237774755
t: 65508228

fitness: 8.59009e+77
```

Run 5:

```
fittest individual found during the 122-th iteration

genotype:
chX: 11110110000100010001000111000100
chY: 11111100001011011111110101100110
chZ: 00000011001001010110001011011110
chT: 00000010000100111110011101000101

phenotype:
x: 4128313796
y: 4230872422
z: 52781790
t: 34858821

fitness: 9.09589e+77
```

Provided these five outputs, the one identified as the optimum is the largest one (run 2, $\sim 9.36 \cdot 10^{77}$). However, there is no way to ensure that this is the global maximum of the function.

Many solution have common patters in their genotypes. All except the 3rd run (which actually produces the smallest solution) have many 1's at the start of variables x and y, and many 0's at the start of z and t. This is possibly a schema featured in highly fit individuals.

4 Conclusions

By means of the basic version of Genetic Algorithms, a solution to the maximum of a very difficult function is proposed ($\sim 9.36 \cdot 10^{77}$ at $x = 4220202170$, $y = 4242441625$, $z = 2064376$ and $t = 147597131$). However, there is no guarantee that this is the actual global maxima. Subsequent runs of the program may produce even better solutions.

References

[1] Luke (George Mason University) Sean. "Essentials of Metaheuristics: A Set of Undergraduate Lecture Notes". In: *Optimization* (2010), pp. 1–220. ISSN: 1389-2576. DOI: 10.1007/s10710-011-9139-0.