
geneticalgs Documentation

Release 1.0

Dmitriy Bobir

Nov 15, 2017

Contents

1	geneticalgs	3
2	Implemented features	5
3	Content description	7
4	Requirements	9
5	Installation	11
6	Running tests	13
7	Documentation	15
8	License	17
9	geneticalgs	19
9.1	geneticalgs package	19
10	Indices and tables	39
	Python Module Index	41

Contents:

CHAPTER 1

geneticalgs

Implementation of standard, migration and diffusion models of genetic algorithms (GA) in `python 3.5`.

Benchmarking was conducted by [COCO platform](#) v15.03.

The project summary may be found in `project_summary.pdf`.

Link to [GitHub](#)

Link to [PyPI](#).

Link to [Read The Docs](#).

Implemented features

- standard, diffusion and migration models
 - with real values (searching for global minimum or maximum of the specified function)
 - with binary encoding combination of some input data
- old population is completely replaced with a new computed one at the end of each generation (generational population model)
- two types of fitness value optimization
 - minimization
 - maximization
- three parent selection types
 - *roulette wheel selection*
 - *rank wheel selection*
 - *tournament*
- may be specified mutation probability
- may be specified any amount of random bits to be mutated
- may be specified crossover probability
- different types of crossover
 - single-point
 - two-point
 - multiple point up to uniform crossover
- elitism may be turned on/off (the best individual may migrate to the next generation)

Content description

- **/geneticalgs/** contains source codes
- **/docs/** contains [sphinx](#) source codes
- **/2.7/** contains files converted from `python 3.5` to `python 2.7` using [3to2 module](#) as [COCO platform](#) used in benchmarking supports only this version of python. These files (not installed package `geneticalgs`) are used in benchmarking. Must be copied in the directory with `my_experiment.py` or `my_timing.py`.
- **/2.7/benchmark/** contains the following files:
 - `my_experiment.py` is used for running benchmarking. Read more [here](#).
 - `my_timing.py` is used for time complexity measurements. It has the same run conditions as the previous file.
 - `pproc.py` is a modified file from COCO platform distribution that must be copied to `bbob.v15.03/python/bbob_pproc/` in order to post-process measured data of migration GA (other models don't need it). It is necessary due to unexpected format of records in case of migration GA.
- **/benchmarking/** contains measured results and the appropriate plots of benchmarking.
- **/time_complexity/** contains time results measured using `my_timing.py`.
- **/examples/** contains examples of using the implemented genetic algorithms.
- **/tests/** contains [pytest](#) tests

CHAPTER 4

Requirements

- python 3.5+
- NumPy
- bitstring
- sphinx for documentation
- pytest for tests

CHAPTER 5

Installation

Install package by typing the command

```
python -m pip install geneticalgs
```

If you have problems installing NumPy it is **strongly** recommended to use [Anaconda](#).

CHAPTER 6

Running tests

You may run tests by typing from the package directory

```
python setup.py test
```


CHAPTER 7

Documentation

Go to the package directory and then to `docs/` and type

```
pip install -r requirements.txt
```

Then type the following command in order to generate documentation in HTML

```
make html
```

And run doctest

```
make doctest
```


CHAPTER 8

License

Licensed under [Apache License Version 2.0](#).

9.1 geneticalgs package

9.1.1 Submodules

9.1.2 geneticalgs.standard_ga module

class `geneticalgs.standard_ga.IndividualGA(chromosome, fitness_val)`

Bases: `object`

The class represents an individual of population in GA.

chromosome

float, list – A chromosome represented a solution. The solution may be binary encoded in chromosome or be a float or a list of floats in case of dealing with real value solutions. The list contains only positions of bit 1 (according to input data list) in case of binary encoded solution.

fitness_val

float, int – Fitness value of the given chromosome.

class `geneticalgs.standard_ga.StandardGA(fitness_func=None, optim='max', selection='rank', mut_prob=0.05, mut_type=1, cross_prob=0.95, cross_type=1, elitism=True, tournament_size=None)`

Bases: `object`

This class implements the base functionality of genetic algorithms and must be inherited. In other words, the class doesn't provide functionality of genetic algorithms by itself. This class is inherited by RealGA and BinaryGA classes in the current implementation.

fitness_func

function – This function must compute fitness value of a single chromosome. Function parameters depend on the implemented subclasses of this class.

optim

str – What this genetic algorithm must do with fitness value: maximize or minimize. May be ‘min’ or ‘max’. Default is “max”.

selection

str – Parent selection type. May be “rank” (Rank Wheel Selection), “roulette” (Roulette Wheel Selection) or “tournament”. Default is “rank”.

tournament_size

int – Defines the size of tournament in case of ‘selection’ == ‘tournament’. Default is None.

mut_prob

float – Probability of mutation. Recommended values are 0.5-1%. Default is 0.5% (0.05).

mut_type

int – This parameter defines how many chromosome bits will be mutated. Default is 1.

cross_prob

float – Probability of crossover. Recommended values are 80-95%. Default is 95% (0.95).

cross_type

int – This parameter defines crossover type. The following types are allowed: single point (1), two point (2) and multiple point ($2 < cross_type$). The extreme case of multiple point crossover is uniform one ($cross_type == all_bits$). The specified number of bits ($cross_type$) are crossed in case of multiple point crossover. Default is 1.

elitism

True, False – Elitism on/off. Default is True.

_check_common_parameters()

This method verifies common input parameters of a genetic algorithm.

_check_init_random_population(*args)

TO BE REIMPLEMENTED IN SUBCLASSES.

This method verifies the input parameters of a random initialization.

_compute_fitness(chromosome)

TO BE REIMPLEMENTED IN SUBCLASSES.

This method computes fitness value of the given chromosome.

Parameters *chromosome* (*float, list*) – A chromosome of genetic algorithm. Defined fitness function (self.fitness_func) must deal with such chromosomes.

Returns fitness value of the given chromosome

_compute_rank_wheel_sum(population_size)

The method returns sum of a wheel that is necessary in parent selection process in case of “rank” selection type.

Parameters *population_size* (*int*) – Size of a population.

Returns sum of the wheel for the given population size

_conduct_tournament(population, size)

Conducts a tournament of the given size within the specified population. The population must be sorted by chromosome’s fitness value the following way: the last population elements are the best.

Parameters

- **population** (*list*) – All possible competitors. Size of the population must be at least 2. Population element is an IndividualGA object.

- **size** (*int*) – Size of a tournament. It will be set to the whole population, if it is greater than the given population size.

Returns indices of a winner of the current tournament and the second best participant

Return type winners (*int*, *int*)

`_cross` (*parent1*, *parent2*)

This method crosses over the two given chromosomes (parents). The first parent is a target chromosome that means its bits will be replaced with bits of the second parent (source chromosome) with the specified crossover probability.

Parameters

- **parent1** (*float*, *list*) – Target chromosome. May be a float or a list of floats, or a binary encoded combination of the original data list (*self.data*) of the first parent.
- **parent2** (*float*, *list*) – Source chromosome. May be a float or a list of floats, or a binary encoded combination of the original data list (*self.data*) of the second parent.

Returns a chromosome (a binary representation, a float or a list of floats) created by the crossover of the two given parents

Return type child (*list*, *float*)

`_generate_random_population` (**args*)

TO BE REIMPLEMENTED IN SUBCLASSES.

This method generates new random population by the given input parameters.

`_invert_bit` (*chromosome*, *bit_num*)

TO BE REIMPLEMENTED IN SUBCLASSES. This method mutates the appropriate bits of the chromosome from *bit_num* with the specified mutation probability.

Parameters

- **chromosome** (*list*, *float*) – A chromosome of population (chromosome without its fitness value).
- **bit_num** (*list*) – List of bits' numbers to invert.

Returns mutated chromosome

`_mutate` (*chromosome*)

This method mutates (inverses bits) the given chromosome.

Parameters **chromosome** (*float*, *list*) – a float or a list of floats, or a binary encoded combination of the original data list (it contains positions of bit 1 according to *self.data*).

Returns

mutated chromosome as float, list of floats or binary representation (any of the mentioned representations with inverted bits depending on subclass)

`_random_diff` (*stop*, *n*, *start=0*)

Creates a list of 'n' different random integer numbers within the interval (start, stop) ('start' included).

Parameters

- **start** (*int*) – Start value of an interval (included). Default is 0.
- **stop** (*int*) – End value of an interval (excluded).
- **n** (*int*) – How many different random numbers must be generated.

Returns list of different random integer values from the given interval ('start' included)

`_replace_bits` (*source*, *target*, *start*, *stop*)

TO BE REIMPLEMENTED IN SUBCLASSES. Replace target bits with source bits in interval (*start*, *stop*) (both included) with the specified crossover probability. This interval represents positions of bits to replace (minimum start point is 0 and maximum end point is *self._bin_length - 1*).

Parameters

- **`source`** (*list*) – Values in source are used as replacement for target.
- **`target`** (*list*) – Values in target are replaced with values in source.
- **`start`** (*int*) – Start point of an interval (included).
- **`stop`** (*int*) – End point of an interval (included).

Returns target with replaced bits with source one in the interval (*start*, *stop*) (both included)

`_select_parents` (*population*, *wheel_sum=None*)

Selects parents from the given population.

Parameters

- **`population`** (*list*) – Current population from which parents will be selected. Population element is an *IndividualGA* object.
- **`wheel_sum`** (*float*) – Sum of values on a wheel (different for “roulette” and “rank”).

Returns selected parents

Return type parents (*IndividualGA*, *IndividualGA*)

`_sort_population` ()

Sorts *self.population* according to *self.optim* (“min” or “max”) in such way that the last element of the population in both cases is the chromosome with the best fitness value.

`_update_solution` (*chromosome*, *fitness_val*)

Updates current best solution if the given one is better.

Parameters

- **`chromosome`** (*float*, *list*) – Chromosome of a population (binary encoded, float or list of floats).
- **`fitness_val`** (*float*, *int*) – Fitness value of the given chromosome.

`best_solution`

Returns tuple in the following form – (best chromosome, its fitness value).

Returns tuple with the currently best found chromosome and its fitness value.

`extend_population` (*elem_list*)

DOES NOT WORK WITH DIFFUSION GENETIC ALGORITHM.

Extends a current population with the new elements. Be careful with type of elements in *elem_list*: they must have the same type as elements of a current population, e.g. *IndividualGA* objects with the *appropriate* chromosome representation (binary encoded for *BinaryGA*, a float or a list of floats for *RealGA*).

Parameters **`elem_list`** (*list*) – New elements of the same type (including chromosome representation) as in the current population.

`init_population` (*chromosomes*, *interval=None*)

Initializes a population with the given chromosomes (binary encoded, float or a list of floats). The fitness values of these chromosomes will be computed by a specified fitness function.

It is recommended to have an amount of chromosomes equal to some squared number (9, 16, 100, 625 etc.) in case of diffusion model of GA. Otherwise some last chromosomes will be lost as the current implementation supports only square arrays of diffusion model.

Parameters

- **chromosomes** (*list*) – Chromosomes of a new population. A single chromosome in case of binary GA is represented as a list of bits' positions with value 1 in the following way: LSB (least significant bit) has position ($\text{len}(\text{self.data}) - 1$) and MSB (most significant bit) has position 0. If it is a GA on real values, a chromosome is represented as a float or a list of floats in case of multiple dimensions. Size of *chromosomes* list must be at least 4.
- **interval** (*tuple*) – An interval in which we are searching the best solution. Must be specified in case of RealGA.

run (*max_generation*)

Starts a standard GA (RealGA or BinaryGA). The algorithm performs *max_generation* generations and then stops. Old population is completely replaced with a new computed one at the end of each generation.

Parameters *max_generation* (*int*) – Maximum number of GA generations.

Returns List of average fitness values for each generation (including original population)

Return type *fitness_progress* (list)

9.1.3 geneticalgs.real_ga module

```
class geneticalgs.real_ga.RealGA (fitness_func=None,      optim='max',      selection='rank',
                                mut_prob=0.05, mut_type=1, cross_prob=0.95, cross_type=1,
                                elitism=True, tournament_size=None)
```

Bases: *geneticalgs.standard_ga.StandardGA*

This class realizes GA over the real values. In other words, it tries to find global minimum or global maximum (depends on the settings) of a given fitness function.

fitness_func

function – This function must compute fitness value of a single chromosome. Function parameters depend on the implemented subclasses of this class.

optim

str – What this genetic algorithm must do with fitness value: maximize or minimize. May be 'min' or 'max'. Default is "max".

selection

str – Parent selection type. May be "rank" (Rank Wheel Selection), "roulette" (Roulette Wheel Selection) or "tournament". Default is "rank".

tournament_size

int – Defines the size of tournament in case of 'selection' == 'tournament'. Default is None.

mut_prob

float – Probability of mutation. Recommended values are 0.5-1%. Default is 0.5% (0.05).

mut_type

int – This parameter defines how many chromosome bits will be mutated. Default is 1.

cross_prob

float – Probability of crossover. Recommended values are 80-95%. Default is 95% (0.95).

cross_type

int – This parameter defines crossover type. The following types are allowed: single point (1), two point

(2) and multiple point ($2 < cross_type$). The extreme case of multiple point crossover is uniform one ($cross_type == all_bits$). The specified number of bits ($cross_type$) are crossed in case of multiple point crossover. Default is 1.

elitism

True, False – Elitism on/off. Default is True.

You may initialize instance of this class the following way

```
from geneticalgs import RealGA
import math

# define some function whose global minimum or maximum we are searching for
# this function takes as input one-dimensional number
def fitness_function(x):
    # the same function is used in examples
    return abs(x*(math.sin(x/11)/5 + math.sin(x/110)))

# initialize standard real GA with fitness maximization by default
gen_alg = RealGA(fitness_function)
# initialize random one-dimensional population of size 20 within interval (0,
↪1000)
gen_alg.init_random_population(20, 1, (0, 1000))
```

Then you may start computation by `gen_alg.run(number_of_generations)` and obtain the currently best found solution by `gen_alg.best_solution`.

`_adjust_to_interval (var)`

This method replaces NaN, inf, -inf in *var* by `numpy.nan_to_num()` and then returns *var* if it is within the specified interval. Otherwise returns lower bound of the interval if (*var* < lower bound) or upper bound of the interval if (*var* > upper bound).

Parameters *var* (*list*, *float*) – A float or a list of floats to adjust to the specified interval.

Returns adjusted input parameter

`_check_init_random_population (size, dim, interval)`

This method verifies the input parameters of a random initialization.

Parameters

- **size** (*int*) – Size of a new population.
- **dim** (*int*) – Amount of space dimensions.
- **interval** (*tuple*) – The generated numbers of each dimension will be within this interval (start point included, end point excluded). Both end points must be *different* integer values.

`_check_parameters ()`**`_compute_fitness (chromosome)`**

This method computes fitness value of the given chromosome.

Parameters *chromosome* (*float*, *list*) – A chromosome of genetic algorithm. May be a single float or a list of floats in case of multiple dimensions. Defined fitness function (*self.fitness_func*) must deal with this chromosome representation.

Returns fitness value of the given chromosome

`_generate_random_population (size, dim, interval)`

This method generates a new random population by the given input parameters.

Parameters

- **size** (*int*) – Size of a new population.
- **dim** (*int*) – Amount of space dimensions.
- **interval** (*tuple*) – The generated numbers of each dimension will be within this interval (start point included, end point excluded).

Returns Array rows represent chromosomes. Number of columns is specified with *dim* parameter.

Return type `array` (`numpy.array`)

`_get_chromosome_return_value` (*chromosome*)

This method returns a vector (chromosome as a list of floats) or a single float depending on number of elements in the given chromosome.

Parameters **chromosome** (*list*) – This list contains a single float or represents a vector of floats in case of multiple dimensions.

Returns *chromosome[0]* iff there is only 1 element in the list, otherwise *chromosome*

`_get_mut_bit_offset` ()

Returns bit number (from left (index 0) to the right) in 32- or 64-bit big-endian floating point binary representation (IEEE 754) from which a mantissa begins. It is necessary because this real GA implementation mutates only mantissa bits (mutation of exponent changes a float number the undesired fast and unexpected way).

`_invert_bit` (*chromosome*, *bit_num*)

This method mutates the appropriate bits of the chromosome from *bit_num* with the specified mutation probability. The method mutates *bit_num*'s bits of all floats in a list represented chromosome in case of multiple dimensions.

Parameters

- **chromosome** (*float*, *list*) – A single float or a list of floats in case of multiple dimensions.
- **bit_num** (*list*) – List of bits' numbers to invert.

Returns mutated chromosome (float, list)

`_is_chromosome_list` (*chromosome*)

This method returns True iff chromosome is a list (even list of just 1 element), otherwise False.

Parameters **chromosome** (*float*, *list*) – A chromosome of GA population. May be float or a list of floats in case of multiple dimensions.

Returns True iff the given chromosome is a list (even a list of just 1 element), otherwise False.

`_replace_bits` (*source*, *target*, *start*, *stop*)

Replaces target bits with source bits in interval (start, stop) (both included) with the specified crossover probability. This interval represents positions of bits to replace (minimum start point is 0 and maximum end point is *self._bin_length - 1*).

Parameters

- **source** (*float*, *list*) – Values in source are used as replacement for target. May be a float or a list of floats in case of multiple dimensions.
- **target** (*float*, *list*) – Values in target are replaced with values in source. May be a float or a list of floats in case of multiple dimensions.
- **start** (*int*) – Start point of interval (included).

- **stop** (*int*) – End point of interval (included).

Returns Target with replaced bits with source one in the interval (start, stop) (both included).

Return type target (*float*, *list*)

init_random_population (*size*, *dim*, *interval*)

Initializes a new random population of the given size with chromosomes' values within the given interval (start point included, end point excluded) with the specified amount of dimensions.

Parameters

- **size** (*int*) – Size of a new random population. Must be at least 2.
- **dim** (*int*) – Amount of space dimensions.
- **interval** (*tuple*) – The generated numbers of each dimension will be within this interval (start point included, end point excluded).

9.1.4 geneticalgs.binary_ga module

```
class geneticalgs.binary_ga.BinaryGA(data=None, fitness_func=None, optim='max',
                                     selection='rank', mut_prob=0.05, mut_type=1,
                                     cross_prob=0.95, cross_type=1, elitism=True, tournament_size=None)
```

Bases: *geneticalgs.standard_ga.StandardGA*

This class realizes GA over a binary encoded input data. In other words, the algorithm tries to find a combination of the input data with the best fitness value.

You may initialize instance of this class the following way

```
from geneticalgs import BinaryGA

# define data whose best combination we are searching for
input_data = [1,2,3,7,-1,-20]

# define a simple fitness function
def fitness_function(chromosome, data):
    # this function searches for the greatest sum of numbers in data
    # chromosome contains positions (from left 0 to right *len(data)-1) of bits 1
    sum = 0
    for bit in chromosome:
        sum += data[bit]

    return sum

# initialize standard binary GA
gen_alg = BinaryGA(input_data, fitness_function)
# initialize random population of size 6
gen_alg.init_random_population(6)
```

Then you may start computation by *gen_alg.run(number_of_generations)* and obtain the currently best found solution by *gen_alg.best_solution*.

_check_init_random_population (*size*)

This method verifies the input parameter of a random initialization.

Parameters **size** (*int*) – Size of a new population to check.

Returns Maximum amount of the input data combinations.

Return type `max_num (int)`

`_check_parameters ()`

`_compute_fitness (chromosome)`

This method computes fitness value of the given chromosome.

Parameters `chromosome (list)` – A binary encoded chromosome of genetic algorithm. Defined fitness function (`self.fitness_func`) must deal with this chromosome representation.

Returns fitness value of the given chromosome

`_generate_random_population (max_num, size)`

This method generates a new random population by the given input parameters.

Parameters

- **`max_num (int)`** – Maximum amount of the input data combinations.
- **`size (int)`** – Size of a new population.

Returns list of integers in interval [1, maxnum) that represents a binary encoded combination.

Return type `population (list)`

`_get_bit_positions (number)`

This method receives a positive decimal integer number and returns positions of bit 1 in its binary representation. However, these positions are transformed the following way: they are mapped on the data list (`self.data`) “as is”. It means that LSB (least significant bit) is mapped on the last position of the data list (e.g. `self._bin_length - 1`), MSB is mapped on the first position of the data list (e.g. 0) and so on.

Parameters `number (int)` – This decimal number represents binary encoded combination of the input data (`self.data`).

Returns list of positions with bit 1 (these positions are mapped on the input data list “as is” and thus, LSB is equal to index (`self._bin_length - 1`) of the input data list).

`_invert_bit (chromosome, bit_num)`

This method mutates the appropriate bits of the given chromosome from `bit_num` with the specified mutation probability.

Parameters

- **`chromosome (list)`** – Binary encoded chromosome (it contains positions of bit 1 according to `self.data`).
- **`bit_num (list)`** – List of bits’ numbers to invert.

Returns mutated chromosome as binary representation of `self.data` (it contains positions of bit 1)

Return type `mutant (list)`

`_replace_bits (source, target, start, stop)`

Replaces target bits with source bits in interval (start, stop) (both included) with the specified crossover probability and returns target. This interval represents positions of bits to replace (minimum start point is 0 and maximum end point is `self._bin_length - 1`).

Parameters

- **`source (list)`** – Values in source are used as replacement for target.
- **`target (list)`** – Values in target are replaced with values in source.
- **`start (int)`** – Start point of interval (included).

- **stop** (*int*) – End point of interval (included).

Returns target with replaced bits in the interval (start, stop) (both included)

init_random_population (*size*)

Initializes a new random population of the given size.

Parameters **size** (*int*) – Size of a new random population. Must be greater than 3 and less than the amount of all possible combinations of the input data.

9.1.5 geneticalgs.diffusion_ga module

class geneticalgs.diffusion_ga.**DiffusionGA** (*instance*)

Bases: `object`

This class implements diffusion model of genetic algorithms. The current implementation supports four neighbours (up, down, left, right) of a currently processed cell. Supports the standard selection types (e.g. “rank”, “roulette”, “tournament”). It’s evident that the maximum tournament size is 4 in this case.

You may initialize instance of this class the following way

```
from geneticalgs import RealGA, DiffusionGA
import math

# define some function whose global minimum or maximum we are searching for
# this function takes as input one-dimensional number
def fitness_function(x):
    # the same function is used in examples
    return abs(x*(math.sin(x/11)/5 + math.sin(x/110)))

# initialize standard real GA with fitness maximization by default
gen_alg = RealGA(fitness_function)
# then initialize diffusion GA using the already initialized real GA instance
dga = DiffusionGA(gen_alg)
# initialize random one-dimensional population of size 20 within interval (0, 1000)
dga.init_random_population(20, 1, (0, 1000))
```

BinaryGA is used the same way. You may start computation by `dga.run(number_of_generations)` and obtain the currently best found solution by `dga.best_solution`.

_compute_diffusion_generation (*chrom_arr*)

This method computes a new generation of a diffusion model of GA.

Parameters **chrom_arr** (*numpy.array*) – Diffusion array of chromosomes (binary encoded, float or a list of floats) of the current generation.

Returns New diffusion arrays of chromosomes and their fitness values of the next generation.

Return type new_chrom_array, new_fitness_arr (*numpy.array*, *numpy.array*)

_construct_diffusion_model (*population*)

Constructs two arrays: first for chromosomes of GA, second for their fitness values. The current implementation supports construction of only 2D square arrays. Thus, an array side is a square root of the given population length. If the calculated square root is a fractional number, it will be truncated that means the last chromosomes in population may not be presented in the constructed arrays.

Parameters **population** (*list*) – List of GA chromosomes. Same as in `self.init_population(new_population)`.

`_find_critical_values` (*fitness_arr*)

Finds 1D or 2D array coordinates of the best and the worst fitness values in the given array. Returns coordinates of the first occurrence of these critical values.

Parameters **fitness_arr** (*numpy.array*) – Array of fitness values.

Returns Coordinates of the best and the worst fitness values as (index_best, index_worst) in 1D or ((row, column), (row, column)) in 2D.

Return type coords_best, coords_worst (*tuple*)

`_get_neighbour` (*row, column*)

The method returns a chromosome selected from the four neighbours (up, down, left, right) of the currently processed cell (specified with the given row and column) according to the selection type (“rank”, “roulette” or “tournament”).

Parameters

- **row** (*int*) – Row of a current cell.
- **column** (*int*) – Column of a current cell.

Returns A chromosome selected from neighbours according to the specified selection type (“rank”, “roulette”, “tournament”).

Return type *chromosome* (binary encoded, *float*, list of floats)

`_init_diffusion_model` (*population*)

This method constructs diffusion model from the given population and then updates the currently best found solution.

Parameters **population** (*list*) – List of GA chromosomes.

`best_solution`

Returns tuple in the following form – (best chromosome, its fitness value).

Returns tuple with the currently best found chromosome and its fitness value.

`init_population` (*new_population*)

Initializes population with the given chromosomes (binary encoded, float or a list of floats) in *new_population*. The fitness values of these chromosomes will be computed by a specified fitness function.

It is recommended to have *new_population* size equal to some squared number (9, 16, 100, 625 etc.) in case of diffusion model of GA. Otherwise some last chromosomes in the given population will be lost as the current implementation supports only square arrays of diffusion model.

Parameters **new_population** (*list*) – A new population of chromosomes of size at least 4. A single chromosome in case of binary GA is represented as a list of bits' positions with value 1 in the following way: LSB (least significant bit) has position (*len(self.data)* - 1) and MSB (most significant bit) has position 0. If it is a GA on real values, an individual is represented as a float or a list of floats in case of multiple dimensions.

`init_random_population` (*size, dim=None, interval=None*)

Initializes a new random population with the given parameters.

Parameters

- **size** (*int*) – A size of new generated population. Must be at least 2 in case of RealGA and at least 4 in case of BinaryGA.
- **dim** (*int, None*) – Amount of space dimensions in case of RealGA.

- **interval** (*tuple*, *None*) – The generated numbers of each dimension will be within this interval (start point included, end point excluded). Must be specified in case of RealGA.

population

Returns the following tuple – (array of chromosomes, array of their fitness values).

Returns Array of chromosomes and another array with their fitness values.

Return type array of chromosomes, array of fitness values (*tuple*)

run (*max_generation*)

Starts a diffusion GA. The algorithm performs *max_generation* generations and then stops. Old population is completely replaced with a new computed one at the end of each generation.

Parameters **max_generation** (*int*) – Maximum number of GA generations.

Returns List of average fitness values for each generation (including original population).

Return type fitness_progress (list)

9.1.6 geneticalgs.migration_ga module

class geneticalgs.migration_ga.**MigrationGA** (*type='binary'*)

Bases: *object*

This class implements migration model of GA, namely island model (not stepping-stone). It works with binary or real GA.

type

str – Type of used genetic algorithms: may be ‘binary’ or ‘real’.

You may initialize instance of this class the following way

```
from geneticalgs import RealGA, MigrationGA
import math

# define some function whose global minimum or maximum we are searching for
# this function takes as input one-dimensional number
def fitness_function(x):
    # the same function is used in examples
    return abs(x*(math.sin(x/11)/5 + math.sin(x/110)))

# initialize two or more standard real GAs with fitness maximization by default
gen_alg1 = RealGA(fitness_function)
gen_alg2 = RealGA(fitness_function)

# initialize random one-dimensional populations of size 10 and 15 within interval
→ (0, 1000)
gen_alg1.init_random_population(10, 1, (0, 1000))
gen_alg2.init_random_population(15, 1, (0, 1000))

# then initialize migration GA using the already initialized standard GA instances
mga = MigrationGA(type='real') # set type of used instances
mga.init_populations([gen_alg1, gen_alg2])
```

Migration model with BinaryGA is used the same way. You may start computation by *mga.run(*args)*.

_check_parameters ()

`_compare_solutions()`

Compares best solutions of the specified GA instances and returns the best solution.

Returns Best solution across all GA instances as (best chromosome, its fitness value).

Return type *best_solution* (tuple)

`init_populations(ga_list)`

This method initializes migration model of GA. Type of optimization ('min' or 'max') will be set to the same value of the first given GA instance. Valid GA instances are RealGA and BinaryGA.

Parameters *ga_list* (list) – List of BinaryGA (or RealGA) instances with already initialized populations.

`run(max_generation, period=1, migrant_num=1, cloning=True, migrate=True)`

Runs a migration model of GA.

Parameters

- **max_generation** (int) – Maximum number of GA generations.
- **period** (int) – How often migration must be performed. Must be less than or equal to *max_generation*.
- **migrant_num** (int) – How many best migrants will travel to all another populations.
- **cloning** (True, False) – Can migrants clone? If False, an original population will not have its migrants after a migration. Otherwise, clones of migrants will remain in their original population after the migration of originals.
- **migrate** (True, False) – Turns on/off migration process. It is useful in case of running GA by only *one* generation so *period* must be also set to 1, but you want to perform migration with period greater than 1 and thus, set migrate initially to False and set it to True when you actually want the algorithm to perform migration. This was used in benchmarking by COCO BBOB platform.

Returns *fitness_progress* contains lists of average fitness value of each generation for each specified GA instance. *best_solution* is the best solution across all GA instances as in form (best chromosome, its fitness value).

Return type *fitness_progress*, *best_solution* (tuple)

You may use this method the standard way

```
avg_fitness_progress, best_solution = mga.run(50, 10, 2)
```

or in more unusual way if you want to get the best found solution for each generation

```
max_generation = 10
for i in range(max_generation):
    # perform migration every four generations
    if i > 0 and i % 3 == 0:
        migrate = True
    else:
        migrate = False

_, best_solution = mga.run(1, 1, 2, cloning=True, migrate=migrate)
```

9.1.7 Module contents

```
class geneticalgs.StandardGA(fitness_func=None, optim='max', selection='rank', mut_prob=0.05,
                             mut_type=1, cross_prob=0.95, cross_type=1, elitism=True, tournament_size=None)
```

Bases: `object`

This class implements the base functionality of genetic algorithms and must be inherited. In other words, the class doesn't provide functionality of genetic algorithms by itself. This class is inherited by `RealGA` and `BinaryGA` classes in the current implementation.

fitness_func

function – This function must compute fitness value of a single chromosome. Function parameters depend on the implemented subclasses of this class.

optim

str – What this genetic algorithm must do with fitness value: maximize or minimize. May be 'min' or 'max'. Default is "max".

selection

str – Parent selection type. May be "rank" (Rank Wheel Selection), "roulette" (Roulette Wheel Selection) or "tournament". Default is "rank".

tournament_size

int – Defines the size of tournament in case of 'selection' == 'tournament'. Default is None.

mut_prob

float – Probability of mutation. Recommended values are 0.5-1%. Default is 0.5% (0.05).

mut_type

int – This parameter defines how many chromosome bits will be mutated. Default is 1.

cross_prob

float – Probability of crossover. Recommended values are 80-95%. Default is 95% (0.95).

cross_type

int – This parameter defines crossover type. The following types are allowed: single point (1), two point (2) and multiple point ($2 < cross_type$). The extreme case of multiple point crossover is uniform one ($cross_type == all_bits$). The specified number of bits ($cross_type$) are crossed in case of multiple point crossover. Default is 1.

elitism

True, False – Elitism on/off. Default is True.

best_solution

Returns tuple in the following form – (best chromosome, its fitness value).

Returns tuple with the currently best found chromosome and its fitness value.

extend_population (*elem_list*)

DOES NOT WORK WITH DIFFUSION GENETIC ALGORITHM.

Extends a current population with the new elements. Be careful with type of elements in *elem_list*: they must have the same type as elements of a current population, e.g. `IndividualGA` objects with the *appropriate* chromosome representation (binary encoded for `BinaryGA`, a float or a list of floats for `RealGA`).

Parameters *elem_list* (*list*) – New elements of the same type (including chromosome representation) as in the current population.

init_population (*chromosomes, interval=None*)

Initializes a population with the given chromosomes (binary encoded, float or a list of floats). The fitness values of these chromosomes will be computed by a specified fitness function.

It is recommended to have an amount of chromosomes equal to some squared number (9, 16, 100, 625 etc.) in case of diffusion model of GA. Otherwise some last chromosomes will be lost as the current implementation supports only square arrays of diffusion model.

Parameters

- **chromosomes** (*list*) – Chromosomes of a new population. A single chromosome in case of binary GA is represented as a list of bits' positions with value 1 in the following way: LSB (least significant bit) has position ($\text{len}(\text{self.data}) - 1$) and MSB (most significant bit) has position 0. If it is a GA on real values, a chromosome is represented as a float or a list of floats in case of multiple dimensions. Size of *chromosomes* list must be at least 4.
- **interval** (*tuple*) – An interval in which we are searching the best solution. Must be specified in case of RealGA.

run (*max_generation*)

Starts a standard GA (RealGA or BinaryGA). The algorithm performs *max_generation* generations and then stops. Old population is completely replaced with a new computed one at the end of each generation.

Parameters *max_generation* (*int*) – Maximum number of GA generations.

Returns List of average fitness values for each generation (including original population)

Return type *fitness_progress* (*list*)

class `geneticalgs.IndividualGA` (*chromosome, fitness_val*)

Bases: `object`

The class represents an individual of population in GA.

chromosome

float, list – A chromosome represented a solution. The solution may be binary encoded in chromosome or be a float or a list of floats in case of dealing with real value solutions. The list contains only positions of bit 1 (according to input data list) in case of binary encoded solution.

fitness_val

float, int – Fitness value of the given chromosome.

class `geneticalgs.BinaryGA` (*data=None, fitness_func=None, optim='max', selection='rank', mut_prob=0.05, mut_type=1, cross_prob=0.95, cross_type=1, elitism=True, tournament_size=None*)

Bases: `geneticalgs.standard_ga.StandardGA`

This class realizes GA over a binary encoded input data. In other words, the algorithm tries to find a combination of the input data with the best fitness value.

You may initialize instance of this class the following way

```
from geneticalgs import BinaryGA

# define data whose best combination we are searching for
input_data = [1,2,3,7,-1,-20]

# define a simple fitness function
def fitness_function(chromosome, data):
    # this function searches for the greatest sum of numbers in data
    # chromosome contains positions (from left 0 to right *len(data)-1) of bits 1
    sum = 0
    for bit in chromosome:
        sum += data[bit]

    return sum
```

```
# initialize standard binary GA
gen_alg = BinaryGA(input_data, fitness_function)
# initialize random population of size 6
gen_alg.init_random_population(6)
```

Then you may start computation by `gen_alg.run(number_of_generations)` and obtain the currently best found solution by `gen_alg.best_solution`.

init_random_population (*size*)

Initializes a new random population of the given size.

Parameters *size* (*int*) – Size of a new random population. Must be greater than 3 and less than the amount of all possible combinations of the input data.

```
class geneticalgs.RealGA(fitness_func=None, optim='max', selection='rank', mut_prob=0.05,
                        mut_type=1, cross_prob=0.95, cross_type=1, elitism=True, tournament_size=None)
```

Bases: `geneticalgs.standard_ga.StandardGA`

This class realizes GA over the real values. In other words, it tries to find global minimum or global maximum (depends on the settings) of a given fitness function.

fitness_func

function – This function must compute fitness value of a single chromosome. Function parameters depend on the implemented subclasses of this class.

optim

str – What this genetic algorithm must do with fitness value: maximize or minimize. May be ‘min’ or ‘max’. Default is “max”.

selection

str – Parent selection type. May be “rank” (Rank Wheel Selection), “roulette” (Roulette Wheel Selection) or “tournament”. Default is “rank”.

tournament_size

int – Defines the size of tournament in case of ‘selection’ == ‘tournament’. Default is None.

mut_prob

float – Probability of mutation. Recommended values are 0.5-1%. Default is 0.5% (0.05).

mut_type

int – This parameter defines how many chromosome bits will be mutated. Default is 1.

cross_prob

float – Probability of crossover. Recommended values are 80-95%. Default is 95% (0.95).

cross_type

int – This parameter defines crossover type. The following types are allowed: single point (1), two point (2) and multiple point ($2 < \text{cross_type}$). The extreme case of multiple point crossover is uniform one ($\text{cross_type} == \text{all_bits}$). The specified number of bits (*cross_type*) are crossed in case of multiple point crossover. Default is 1.

elitism

True, False – Elitism on/off. Default is True.

You may initialize instance of this class the following way

```
from geneticalgs import RealGA
import math
```

```
# define some function whose global minimum or maximum we are searching for
# this function takes as input one-dimensional number
def fitness_function(x):
    # the same function is used in examples
    return abs(x*(math.sin(x/11)/5 + math.sin(x/110)))

# initialize standard real GA with fitness maximization by default
gen_alg = RealGA(fitness_function)
# initialize random one-dimensional population of size 20 within interval (0,
↪1000)
gen_alg.init_random_population(20, 1, (0, 1000))
```

Then you may start computation by `gen_alg.run(number_of_generations)` and obtain the currently best found solution by `gen_alg.best_solution`.

init_random_population (*size, dim, interval*)

Initializes a new random population of the given size with chromosomes' values within the given interval (start point included, end point excluded) with the specified amount of dimensions.

Parameters

- **size** (*int*) – Size of a new random population. Must be at least 2.
- **dim** (*int*) – Amount of space dimensions.
- **interval** (*tuple*) – The generated numbers of each dimension will be within this interval (start point included, end point excluded).

class `geneticalgs.DiffusionGA` (*instance*)

Bases: `object`

This class implements diffusion model of genetic algorithms. The current implementation supports four neighbours (up, down, left, right) of a currently processed cell. Supports the standard selection types (e.g. “rank”, “roulette”, “tournament”). It's evident that the maximum tournament size is 4 in this case.

You may initialize instance of this class the following way

```
from geneticalgs import RealGA, DiffusionGA
import math

# define some function whose global minimum or maximum we are searching for
# this function takes as input one-dimensional number
def fitness_function(x):
    # the same function is used in examples
    return abs(x*(math.sin(x/11)/5 + math.sin(x/110)))

# initialize standard real GA with fitness maximization by default
gen_alg = RealGA(fitness_function)
# then initialize diffusion GA using the already initialized real GA instance
dga = DiffusionGA(gen_alg)
# initialize random one-dimensional population of size 20 within interval (0,
↪1000)
dga.init_random_population(20, 1, (0, 1000))
```

BinaryGA is used the same way. You may start computation by `dga.run(number_of_generations)` and obtain the currently best found solution by `dga.best_solution`.

best_solution

Returns tuple in the following form – (best chromosome, its fitness value).

Returns tuple with the currently best found chromosome and its fitness value.

init_population (*new_population*)

Initializes population with the given chromosomes (binary encoded, float or a list of floats) in *new_population*. The fitness values of these chromosomes will be computed by a specified fitness function.

It is recommended to have *new_population* size equal to some squared number (9, 16, 100, 625 etc.) in case of diffusion model of GA. Otherwise some last chromosomes in the given population will be lost as the current implementation supports only square arrays of diffusion model.

Parameters *new_population* (*list*) – A new population of chromosomes of size at least 4. A single chromosome in case of binary GA is represented as a list of bits' positions with value 1 in the following way: LSB (least significant bit) has position (*len(self.data)* - 1) and MSB (most significant bit) has position 0. If it is a GA on real values, an individual is represented as a float or a list of floats in case of multiple dimensions.

init_random_population (*size*, *dim=None*, *interval=None*)

Initializes a new random population with the given parameters.

Parameters

- **size** (*int*) – A size of new generated population. Must be at least 2 in case of RealGA and at least 4 in case of BinaryGA.
- **dim** (*int*, *None*) – Amount of space dimensions in case of RealGA.
- **interval** (*tuple*, *None*) – The generated numbers of each dimension will be within this interval (start point included, end point excluded). Must be specified in case of RealGA.

population

Returns the following tuple – (array of chromosomes, array of their fitness values).

Returns Array of chromosomes and another array with their fitness values.

Return type array of chromosomes, array of fitness values (*tuple*)

run (*max_generation*)

Starts a diffusion GA. The algorithm performs *max_generation* generations and then stops. Old population is completely replaced with a new computed one at the end of each generation.

Parameters *max_generation* (*int*) – Maximum number of GA generations.

Returns List of average fitness values for each generation (including original population).

Return type fitness_progress (*list*)

class `geneticalgs.MigrationGA` (*type='binary'*)

Bases: `object`

This class implements migration model of GA, namely island model (not stepping-stone). It works with binary or real GA.

type

str – Type of used genetic algorithms: may be 'binary' or 'real'.

You may initialize instance of this class the following way

```
from geneticalgs import RealGA, MigrationGA
import math

# define some function whose global minimum or maximum we are searching for
# this function takes as input one-dimensional number
def fitness_function(x):
    # the same function is used in examples
    return abs(x*(math.sin(x/11)/5 + math.sin(x/110)))
```



```
# initialize two or more standard real GAs with fitness maximization by default
gen_alg1 = RealGA(fitness_function)
gen_alg2 = RealGA(fitness_function)

# initialize random one-dimensional populations of size 10 and 15 within interval_
↪ (0, 1000)
gen_alg1.init_random_population(10, 1, (0, 1000))
gen_alg2.init_random_population(15, 1, (0, 1000))

# then initialize migration GA using the already initialized standard GA instances
mga = MigrationGA(type='real') # set type of used instances
mga.init_populations([gen_alg1, gen_alg2])
```

Migration model with BinaryGA is used the same way. You may start computation by `mga.run(*args)`.

init_populations (*ga_list*)

This method initializes migration model of GA. Type of optimization ('min' or 'max') will be set to the same value of the first given GA instance. Valid GA instances are RealGA and BinaryGA.

Parameters `ga_list` (*list*) – List of BinaryGA (or RealGA) instances with already initialized populations.

run (*max_generation, period=1, migrant_num=1, cloning=True, migrate=True*)

Runs a migration model of GA.

Parameters

- **max_generation** (*int*) – Maximum number of GA generations.
- **period** (*int*) – How often migration must be performed. Must be less than or equal to *max_generation*.
- **migrant_num** (*int*) – How many best migrants will travel to all another populations.
- **cloning** (*True, False*) – Can migrants clone? If False, an original population will not have its migrants after a migration. Otherwise, clones of migrants will remain in their original population after the migration of originals.
- **migrate** (*True, False*) – Turns on/off migration process. It is useful in case of running GA by only *one* generation so *period* must be also set to 1, but you want to perform migration with period greater than 1 and thus, set migrate initially to False and set it to True when you actually want the algorithm to perform migration. This was used in benchmarking by COCO BBOB platform.

Returns *fitness_progress* contains lists of average fitness value of each generation for each specified GA instance. *best_solution* is the best solution across all GA instances as in form (best chromosome, its fitness value).

Return type *fitness_progress, best_solution* (tuple)

You may use this method the standard way

```
avg_fitness_progress, best_solution = mga.run(50, 10, 2)
```

or in more unusual way if you want to get the best found solution for each generation

```
max_generation = 10
for i in range(max_generation):
    # perform migration every four generations
    if i > 0 and i % 3 == 0:
```

```
        migrate = True
    else:
        migrate = False

_, best_solution = mga.run(1, 1, 2, cloning=True, migrate=migrate)
```

CHAPTER 10

Indices and tables

- `genindex`
- `modindex`
- `search`

g

- `geneticalgs`, [32](#)
- `geneticalgs.binary_ga`, [26](#)
- `geneticalgs.diffusion_ga`, [28](#)
- `geneticalgs.migration_ga`, [30](#)
- `geneticalgs.real_ga`, [23](#)
- `geneticalgs.standard_ga`, [19](#)

Symbols

<code>_adjust_to_interval()</code>	(geneticalgs.real_ga.RealGA method), 24
<code>_check_common_parameters()</code>	(geneticalgs.standard_ga.StandardGA method), 20
<code>_check_init_random_population()</code>	(geneticalgs.binary_ga.BinaryGA method), 26
<code>_check_init_random_population()</code>	(geneticalgs.real_ga.RealGA method), 24
<code>_check_init_random_population()</code>	(geneticalgs.standard_ga.StandardGA method), 20
<code>_check_parameters()</code>	(geneticalgs.binary_ga.BinaryGA method), 27
<code>_check_parameters()</code>	(geneticalgs.migration_ga.MigrationGA method), 30
<code>_check_parameters()</code>	(geneticalgs.real_ga.RealGA method), 24
<code>_compare_solutions()</code>	(geneticalgs.migration_ga.MigrationGA method), 30
<code>_compute_diffusion_generation()</code>	(geneticalgs.diffusion_ga.DiffusionGA method), 28
<code>_compute_fitness()</code>	(geneticalgs.binary_ga.BinaryGA method), 27
<code>_compute_fitness()</code>	(geneticalgs.real_ga.RealGA method), 24
<code>_compute_fitness()</code>	(geneticalgs.standard_ga.StandardGA method), 20
<code>_compute_rank_wheel_sum()</code>	(geneticalgs.standard_ga.StandardGA method), 20
<code>_conduct_tournament()</code>	(geneticalgs.standard_ga.StandardGA method), 20
<code>_construct_diffusion_model()</code>	(geneticalgs.diffusion_ga.DiffusionGA method), 28
<code>_cross()</code>	(geneticalgs.standard_ga.StandardGA method), 21
<code>_find_critical_values()</code>	(geneticalgs.diffusion_ga.DiffusionGA method), 28
<code>_generate_random_population()</code>	(geneticalgs.binary_ga.BinaryGA method), 27
<code>_generate_random_population()</code>	(geneticalgs.real_ga.RealGA method), 24
<code>_generate_random_population()</code>	(geneticalgs.standard_ga.StandardGA method), 21
<code>_get_bit_positions()</code>	(geneticalgs.binary_ga.BinaryGA method), 27
<code>_get_chromosome_return_value()</code>	(geneticalgs.real_ga.RealGA method), 25
<code>_get_mut_bit_offset()</code>	(geneticalgs.real_ga.RealGA method), 25
<code>_get_neighbour()</code>	(geneticalgs.diffusion_ga.DiffusionGA method), 29
<code>_init_diffusion_model()</code>	(geneticalgs.diffusion_ga.DiffusionGA method), 29
<code>_invert_bit()</code>	(geneticalgs.binary_ga.BinaryGA method), 27
<code>_invert_bit()</code>	(geneticalgs.real_ga.RealGA method), 25
<code>_invert_bit()</code>	(geneticalgs.standard_ga.StandardGA method), 21
<code>_is_chromosome_list()</code>	(geneticalgs.real_ga.RealGA method), 25
<code>_mutate()</code>	(geneticalgs.standard_ga.StandardGA method), 21
<code>_random_diff()</code>	(geneticalgs.standard_ga.StandardGA method), 21
<code>_replace_bits()</code>	(geneticalgs.binary_ga.BinaryGA method), 27
<code>_replace_bits()</code>	(geneticalgs.real_ga.RealGA method), 25

`_replace_bits()` (geneticalgs.standard_ga.StandardGA method), 21
`_select_parents()` (geneticalgs.standard_ga.StandardGA method), 22
`_sort_population()` (geneticalgs.standard_ga.StandardGA method), 22
`_update_solution()` (geneticalgs.standard_ga.StandardGA method), 22

B

`best_solution` (geneticalgs.diffusion_ga.DiffusionGA attribute), 29
`best_solution` (geneticalgs.DiffusionGA attribute), 35
`best_solution` (geneticalgs.standard_ga.StandardGA attribute), 22
`best_solution` (geneticalgs.StandardGA attribute), 32
BinaryGA (class in geneticalgs), 33
BinaryGA (class in geneticalgs.binary_ga), 26

C

`chromosome` (geneticalgs.IndividualGA attribute), 33
`chromosome` (geneticalgs.standard_ga.IndividualGA attribute), 19
`cross_prob` (geneticalgs.real_ga.RealGA attribute), 23
`cross_prob` (geneticalgs.RealGA attribute), 34
`cross_prob` (geneticalgs.standard_ga.StandardGA attribute), 20
`cross_prob` (geneticalgs.StandardGA attribute), 32
`cross_type` (geneticalgs.real_ga.RealGA attribute), 23
`cross_type` (geneticalgs.RealGA attribute), 34
`cross_type` (geneticalgs.standard_ga.StandardGA attribute), 20
`cross_type` (geneticalgs.StandardGA attribute), 32

D

DiffusionGA (class in geneticalgs), 35
DiffusionGA (class in geneticalgs.diffusion_ga), 28

E

`elitism` (geneticalgs.real_ga.RealGA attribute), 24
`elitism` (geneticalgs.RealGA attribute), 34
`elitism` (geneticalgs.standard_ga.StandardGA attribute), 20
`elitism` (geneticalgs.StandardGA attribute), 32
`extend_population()` (geneticalgs.standard_ga.StandardGA method), 22
`extend_population()` (geneticalgs.StandardGA method), 32

F

`fitness_func` (geneticalgs.real_ga.RealGA attribute), 23
`fitness_func` (geneticalgs.RealGA attribute), 34

`fitness_func` (geneticalgs.standard_ga.StandardGA attribute), 19
`fitness_func` (geneticalgs.StandardGA attribute), 32
`fitness_val` (geneticalgs.IndividualGA attribute), 33
`fitness_val` (geneticalgs.standard_ga.IndividualGA attribute), 19

G

geneticalgs (module), 32
geneticalgs.binary_ga (module), 26
geneticalgs.diffusion_ga (module), 28
geneticalgs.migration_ga (module), 30
geneticalgs.real_ga (module), 23
geneticalgs.standard_ga (module), 19

I

IndividualGA (class in geneticalgs), 33
IndividualGA (class in geneticalgs.standard_ga), 19
`init_population()` (geneticalgs.diffusion_ga.DiffusionGA method), 29
`init_population()` (geneticalgs.DiffusionGA method), 35
`init_population()` (geneticalgs.standard_ga.StandardGA method), 22
`init_population()` (geneticalgs.StandardGA method), 32
`init_populations()` (geneticalgs.migration_ga.MigrationGA method), 31
`init_populations()` (geneticalgs.MigrationGA method), 37
`init_random_population()` (geneticalgs.binary_ga.BinaryGA method), 28
`init_random_population()` (geneticalgs.BinaryGA method), 34
`init_random_population()` (geneticalgs.diffusion_ga.DiffusionGA method), 29
`init_random_population()` (geneticalgs.DiffusionGA method), 36
`init_random_population()` (geneticalgs.real_ga.RealGA method), 26
`init_random_population()` (geneticalgs.RealGA method), 35

M

MigrationGA (class in geneticalgs), 36
MigrationGA (class in geneticalgs.migration_ga), 30
`mut_prob` (geneticalgs.real_ga.RealGA attribute), 23
`mut_prob` (geneticalgs.RealGA attribute), 34
`mut_prob` (geneticalgs.standard_ga.StandardGA attribute), 20
`mut_prob` (geneticalgs.StandardGA attribute), 32
`mut_type` (geneticalgs.real_ga.RealGA attribute), 23
`mut_type` (geneticalgs.RealGA attribute), 34
`mut_type` (geneticalgs.standard_ga.StandardGA attribute), 20

mut_type (geneticalgs.StandardGA attribute), [32](#)

O

optim (geneticalgs.real_ga.RealGA attribute), [23](#)

optim (geneticalgs.RealGA attribute), [34](#)

optim (geneticalgs.standard_ga.StandardGA attribute), [19](#)

optim (geneticalgs.StandardGA attribute), [32](#)

P

population (geneticalgs.diffusion_ga.DiffusionGA attribute), [30](#)

population (geneticalgs.DiffusionGA attribute), [36](#)

R

RealGA (class in geneticalgs), [34](#)

RealGA (class in geneticalgs.real_ga), [23](#)

run() (geneticalgs.diffusion_ga.DiffusionGA method), [30](#)

run() (geneticalgs.DiffusionGA method), [36](#)

run() (geneticalgs.migration_ga.MigrationGA method), [31](#)

run() (geneticalgs.MigrationGA method), [37](#)

run() (geneticalgs.standard_ga.StandardGA method), [23](#)

run() (geneticalgs.StandardGA method), [33](#)

S

selection (geneticalgs.real_ga.RealGA attribute), [23](#)

selection (geneticalgs.RealGA attribute), [34](#)

selection (geneticalgs.standard_ga.StandardGA attribute), [20](#)

selection (geneticalgs.StandardGA attribute), [32](#)

StandardGA (class in geneticalgs), [32](#)

StandardGA (class in geneticalgs.standard_ga), [19](#)

T

tournament_size (geneticalgs.real_ga.RealGA attribute), [23](#)

tournament_size (geneticalgs.RealGA attribute), [34](#)

tournament_size (geneticalgs.standard_ga.StandardGA attribute), [20](#)

tournament_size (geneticalgs.StandardGA attribute), [32](#)

type (geneticalgs.migration_ga.MigrationGA attribute), [30](#)

type (geneticalgs.MigrationGA attribute), [36](#)