PyGenAlgo

Release 00.00.01

Michalis Vrettas, PhD

Nov 08, 2024

CONTENTS:

1 pygenalgo **1**

1.1 pygenalgo package 1

2 Indices and tables **23**

Python Module Index **25**

Index **27**

CHAPTER

ONE

PYGENALGO

1.1 pygenalgo package

1.1.1 Subpackages

pygenalgo.engines package

Submodules

pygenalgo.engines.auxiliary module

class pygenalgo.engines.auxiliary.**SubPopulation**(*pop_id: int*, *population: list = <factory>*)

Bases: `object`

Auxiliary class container used in the IslandModelGA to hold all the subpopulations (one on each island).

property id: int

Accessor (getter) of the id parameter.

Returns
the id value.

pop_id: int

population: list

pygenalgo.engines.auxiliary.**apply_corrections**(*input_population: list[*Chromosome*]*, *fit_func: Callable | None = None*) → int

Check the population for invalid genes and correct them by applying directly the random method. It is assumed that the random method of the Gene is always returning a 'valid' value for the Gene. After that, we need to reevaluate the chromosome to update its fitness.

Parameters
input_population – List(Chromosome) the population

we want to apply corrections (if applicable).

Parameters
fit_func – callable fitness function.

Returns
the total number of corrected genes in the population.

pygenalgo.engines.auxiliary.**avg_hamming_dist**(*input_population: list[*Chromosome*]*) → float

Computes the average Hamming distance of a population. We use this to measure the similarity in the population of chromosomes.

Parameters
input_population – List(Chromosome) the population we want

to compute the average Hamming distance.

Returns
(float) the total number of differences, in the genes,

divided by the total number of genes compared.

pygenalgo.engines.generic_ga module

class pygenalgo.engines.generic_ga.**GenericGA**(*initial_pop: list[*Chromosome*]*, *fit_func: Callable*, *select_op:* SelectionOperator | *None = None*, *mutate_op:* MutationOperator | *None = None*, *crossx_op:* CrossoverOperator | *None = None*)

Bases: `object`

Description:

Generic GA class models the interface of a specific genetic algorithm model (or engine). It provides the common variables and functionality that all GA models should share.

MAX_CPUs = 4

best_chromosome() → *Chromosome*

Auxiliary method that returns the chromosome with the highest fitness value. Safeguarded with ignoring NaNs.

Returns
Return the chromosome with the highest fitness.

crossover_mutate(*input_population: list[*Chromosome*]*) → None

This is an auxiliary method that combines the crossover and mutation operations in one call. Since these operations happen in place the 'input_population' will be modified directly.

This method should be called AFTER the selection of the parents that have been selected for breeding.

Parameters
input_population – this is the population that we will apply

the two genetic operators.

property crossover_op: *CrossoverOperator*

Accessor method that returns the crossover operator reference.

Returns
the CrossoverOperator.

evaluate_fitness(**args*, ***kwargs*)

This method evaluates all the chromosomes' of an input population with a custom fitness function. After updating all the chromosomes with their fitness, the method should return the average statistics of mean and std of the population fitness.

fitness_func

individual_fitness(*index: int*) → float

Get the fitness value of an individual member of the population.

Parameters
index – Position of the individual in the population.

Returns
The fitness value (float).

property mutate_op: *MutationOperator*

Accessor method that returns the mutation operator reference.

Returns
the MutationOperator.

population

population_fitness() → list[float]

Get the fitness values of all the population.

Returns
A list with all the fitness values.

rng_GA = Generator(PCG64) at 0x7FB7AE9804A0

run(**args*, ***kwargs*)

Main method of the Generic GA class, that implements the evolutionary routine.

property select_op: *SelectionOperator*

Accessor method that returns the selection operator reference.

Returns
the SelectionOperator.

property stats: dict

Accessor method that returns the 'stats' dictionary.

Returns
the dictionary with the statistics from the run.

pygenalgo.engines.island_model_ga module

class pygenalgo.engines.island_model_ga.**IslandModelGA**(*num_islands: int*, *migrate_op:* ClockwiseMigration | *None* = *None*, ***kwargs*)

Bases: *GenericGA*

Description:

In Island Model GA we run in parallel a number of "islands", each one evolving its own (sub)-population. Optionally we can allow "migration", among the best individuals from each island.

evaluate_fitness(*in_population: list[~pygenalgo.genome.chromosome.Chromosome]) -> (<class 'float'>, <class 'float'>)*

Evaluate all the chromosomes of the input population list with the custom fitness function. After updating all the chromosomes with their fitness, the method returns the average statistics mean/std.

Parameters
in_population – (list) The population of Chromosomes that we

want to evaluate their fitness.

Returns
mean(fitness), std(fitness).

classmethod evolve_population(*island:* SubPopulation, *eval_fitness: Callable*, *epochs: int*, *crs_op:* CrossoverOperator, *mut_op:* MutationOperator, *sel_op:* SelectionOperator, *rnd_gen*, *f_tol: float | None = None*, *correction: bool = False*, *elitism: bool = True*)

This method is called to evolve each subpopulation independently. It is defined as 'classmethod' because we need access to the fitness function of the object. The input parameters have identical meaning with the ones from run().

property migrate_op: *MigrationOperator*

Accessor method that returns the migration operator reference.

Returns
the MigrationOperator.

num_islands

print_migration_stats() → None

Print the migration operators stats.

Returns
None.

run(*epochs: int = 1000*, *correction: bool = False*, *elitism: bool = True*, *f_tol: float | None = None*, *allow_migration: bool = False*, *n_periods: int = 10*, *verbose: bool = False*) → None

Main method of the IslandModelGA class, that implements the evolutionary routine.

Parameters

- **epochs** – (int) maximum number of iterations in the evolution process.
- **correction** – (bool) flag that if set to 'True' will check the validity of

the population (at the gene level) and attempt to correct the genome by calling the random() method of the flawed gene.

Parameters
elitism – (bool) flag that defines elitism. If 'True' then the chromosome

with the higher fitness will always be copied to the next generation (unaltered).

Parameters
f_tol – (float) tolerance in the difference between the average values of two

consecutive populations. It is used to determine the convergence of the population. If this value is None (default) the algorithm will terminate using the epochs value.

Parameters
allow_migration – (bool) flag that if set to 'True' will allow the migration

of the best individuals among the different islands.

Parameters
n_periods – (int) the number of times that we will break the main evolution

to allow for chromosomes to migrate. NB: This setting is active only when the option allow_migration == True. Otherwise, is ignored.

Parameters
verbose – (bool) if 'True' it will display periodically information about the

current stats of the subpopulations. NB: This setting is active only when the option allow_migration == True. Otherwise, is ignored.

Returns
None.

pygenalgo.engines.standard_ga module

class pygenalgo.engines.standard_ga.**StandardGA**(***kwargs*)

Bases: *GenericGA*

Description:

StandardGA model provides a basic implementation of the "GenericGA", which at each iteration (epoch) replaces the whole population using the genetic operators (crossover and mutation).

evaluate_fitness(*input_population: list[*Chromosome*]*, *parallel: bool = False*) → list[float]

Evaluate all the chromosomes of the input list with the custom fitness function.

Parameters
input_population – (list) The population of Chromosomes that we want to

evaluate their fitness.

Parameters
parallel – (bool) Flag that enables parallel computation of the fitness function.

Returns
a list of the fitness values.

fitness_func

population

print_operator_stats() → None

Print the genetic operators stats.

Returns
None.

run(*epochs: int = 100*, *elitism: bool = True*, *correction: bool = False*, *f_tol: float | None = None*, *parallel: bool = False*, *verbose: bool = False*) → None

Main method of the StandardGA class, that implements the evolutionary routine.

Parameters

- **epochs** – (int) maximum number of iterations in the evolution process.
- **elitism** – (bool) flag that defines elitism. If 'True' then the chromosome

with the higher fitness will always be copied to the next generation (unaltered).

Parameters
correction – (bool) flag that if set to 'True' will check the validity of

the population (at the gene level) and attempt to correct the genome by calling the random() method of the flawed gene.

Parameters
f_tol – (float) tolerance in the difference between the average values of two

consecutive populations. It is used to determine the convergence of the population. If this value is None (default) the algorithm will terminate using the epochs value.

Parameters

- **parallel** – (bool) Flag that enables parallel computation of the fitness function.
- **verbose** – (bool) if 'True' it will display periodically information about

the current average fitness and spread of the population.

Returns
None.

update_stats(*fit_list: list[float]) -> (<class 'float'>, <class 'float'>)*

Update the stats dictionary with the mean/std values of the population fitness values.

Parameters
fit_list – (float) mean fitness value of the population.

Returns
the mean and std of the fitness values.

Module contents

pygenalgo.genome package

Submodules

pygenalgo.genome.chromosome module

class pygenalgo.genome.chromosome.**Chromosome**(*_genome: list = <factory>, _fitness: float = 0.0, _valid: bool = True*)

Bases: `object`

Description:

Implements a dataclass for the Chromosome entity. This class is responsible for holding the individual solution(s), of the optimization problem, during the evolution process.

clone()

Makes a duplicate of the self object.

Returns
a "deep-copy" of the object.

property fitness: float

Accessor of the fitness value of the chromosome.

Returns
the fitness (float) of the genome.

property genome: list[*Gene*]

Accessor of the genome list of the chromosome.

Returns
the list (of Genes) of the chromosome.

hamming_distance(*other*) → int

Compute the "Hamming distance" of the "self" object with the "other" chromosome. In practise it's the number of positions at which the corresponding genes are different.

Parameters
other – (Chromosome) to compare the Hamming distance.

Returns
(int) the distance between the two chromosomes.

is_genome_valid() → bool

Checks the validity of the whole chromosome, by calling individually all genes is_valid method.

In addition, it "double-checks" that all entries in the genome are of type 'Gene'.

Returns
True if ALL genes are valid, else False.

property valid: bool

Accessor (getter) of the validity parameter.

Returns
the valid value.

pygenalgo.genome.gene module

class pygenalgo.genome.gene.**Gene**(*datum: Any*, *func: Callable*, *valid: bool = True*)

Bases: object

Description:

This is the main class that encodes the data of a single Gene in the chromosome. The class encapsulates not only the data, but also the way that this gene can be mutated using a random function. This Gene can be from a single 'bit' to a whole image. This way provides us with flexibility to parameterize the chromosome with different "kinds of genes" each one responsible for a specific function.

clone()

Makes a duplicate of the self object.

Returns
a "deep-copy" of the object.

flip() → None

This method flips the value of the gene data. It is used only dy the FlipMutator operator for problems where the chromosome is represented by a list of bits.

i) 1 -> 0

ii) 0 -> 1

Returns
None.

gaussian() → None

This method adds a random value, drawn from a standard normal distribution x ~ N(0,1) to the current gene data value. It is used mostly from the GaussianMutator method.

Returns
None.

property is_valid: bool

Accessor (getter) of the validity parameter.

Returns
the valid value.

random() → None

This method should be different for each type of Gene. It describes how a specific type of Gene creates a random version of itself. The main idea is that inside the Chromosome, each Gene can represent a very different concept of the problem solution, so its Gene should have its own way to perform random mutation.

This way by calling on the random() method, each Gene will know how to mutate itself without breaking any rules/constraints.

Returns
None.

property value: Any

Accessor (getter) of the data reference.

Returns
the datum value.

Module contents

pygenalgo.operators package

Subpackages

pygenalgo.operators.crossover package

Submodules

pygenalgo.operators.crossover.crossover_operator module

class pygenalgo.operators.crossover.crossover_operator.**CrossoverOperator**(*crossover_probability: float*)

Bases: *GeneticOperator*

Description:

Provides the base class (interface) for a Crossover Operator.

crossover(*parent1:* Chromosome, *parent2:* Chromosome)

Abstract method that "reminds" the user that if they want to create a Crossover Class that inherits from here they should implement a crossover method.

Parameters

- **parent1** – (Chromosome).
- **parent2** – (Chromosome).

Returns
Nothing but raising an error.

pygenalgo.operators.crossover.meta_crossover module

class pygenalgo.operators.crossover.meta_crossover.**MetaCrossover**(*crossover_probability: float = 0.9*)

Bases: *CrossoverOperator*

Description:

Meta-crossover, crosses the chromosomes by applying randomly all other crossovers (one at a time), with equal probability.

property all_counters: dict

Accessor (getter) of the application counter from all the internal crossovers. This is mostly to verify that everything is working as expected.

Returns
a dictionary with the counter calls for all crossover methods.

crossover(*parent1:* Chromosome, *parent2:* Chromosome)

Perform the crossover operation on the two input parent chromosomes, by selecting randomly a predefined method.

Parameters

- **parent1** – (Chromosome).
- **parent2** – (Chromosome).

Returns
child1 and child2 (as Chromosomes).

reset_counter() → None

Sets ALL the counters to 'zero'. We have to override the super().reset_counter() method, because we have to call explicitly the reset_counter on all the internal operators.

Returns
None.

pygenalgo.operators.crossover.mutli_point_crossover module

class pygenalgo.operators.crossover.mutli_point_crossover.**MultiPointCrossover**(*crossover_probability: float = 0.9*, *num_loci: int = 2*)

Bases: *CrossoverOperator*

Description:

Multipoint crossover creates two children chromosomes (offsprings), by taking two parent chromosomes and cutting them at randomly chosen, sites (loci).

It produces faster mixing, compared with single-point crossover.

crossover(*parent1:* Chromosome, *parent2:* Chromosome)

Perform the crossover operation on the two input parent chromosomes, using multiple cutting points (num_loci).

NOTE: the number of loci is held in the '_items' variable.

Parameters

- **parent1** – (Chromosome).
- **parent2** – (Chromosome).

Returns
child1 and child2 (as Chromosomes).

pygenalgo.operators.crossover.order_crossover module

class pygenalgo.operators.crossover.order_crossover.**OrderCrossover**(*crossover_probability: float = 0.9*)

Bases: *CrossoverOperator*

Description:

Order crossover (OX1) creates two children chromosomes, by ensuring that the original genome (from both parents) isn't repeated, thus creating invalid offsprings.

It is used predominantly in combinatorial problems.

crossover(*parent1:* Chromosome, *parent2:* Chromosome)

Perform the crossover operation on the two input parent chromosomes.

Parameters

- **parent1** – (Chromosome).
- **parent2** – (Chromosome).

Returns
child1 and child2 (as Chromosomes).

pygenalgo.operators.crossover.partially_mapped_crossover module

class pygenalgo.operators.crossover.partially_mapped_crossover.**PartiallyMappedCrossover**(*crossover_probabili float = 0.9*)

Bases: *CrossoverOperator*

Description:

Partially Mapped Crossover (PMX) creates two children chromosomes, by ensuring that the original genome (from both parents) isn't repeated, thus creating invalid offsprings.

It is used predominantly in combinatorial problems.

crossover(*parent1:* Chromosome, *parent2:* Chromosome)

Perform the crossover operation on the two input parent chromosomes.

Parameters

- **parent1** – (Chromosome).
- **parent2** – (Chromosome).

Returns
child1 and child2 (as Chromosomes).

pygenalgo.operators.crossover.position_based_crossover module

class pygenalgo.operators.crossover.position_based_crossover.**PositionBasedCrossover**(*crossover_probability: float = 0.9*)

Bases: *CrossoverOperator*

Description:

Position based crossover (POS) creates two children chromosomes, by ensuring that the original genome (from both parents) isn't repeated, thus creating invalid offsprings.

It is used predominantly in combinatorial problems.

crossover(*parent1:* Chromosome, *parent2:* Chromosome)

Perform the crossover operation on the two input parent chromosomes.

Parameters

- **parent1** – (Chromosome).
- **parent2** – (Chromosome).

Returns
child1 and child2 (as Chromosomes).

pygenalgo.operators.crossover.single_point_crossover module

class pygenalgo.operators.crossover.single_point_crossover.**SinglePointCrossover**(*crossover_probability: float = 0.9*)

Bases: *CrossoverOperator*

Description:

Single-point crossover creates two children chromosomes (offsprings), by taking two parent chromosomes and cutting them at some, randomly chosen, site (locus).

It produces very slow mixing, compared with multipoint or uniform crossover.

crossover(*parent1:* Chromosome, *parent2:* Chromosome)

Perform the crossover operation on the two input parent chromosomes.

Parameters

- **parent1** – (Chromosome).
- **parent2** – (Chromosome).

Returns
child1 and child2 (as Chromosomes).

pygenalgo.operators.crossover.uniform_crossover module

class pygenalgo.operators.crossover.uniform_crossover.**UniformCrossover**(*crossover_probability: float = 0.9*)

Bases: *CrossoverOperator*

Description:

Uniform crossover creates two children chromosomes (offsprings), by taking two parent chromosomes and swap their genes in every other location.

It produces fast mixing, compared with single-point crossover.

crossover(*parent1:* Chromosome, *parent2:* Chromosome)

Perform the crossover operation on the two input parent chromosomes.

Parameters

- **parent1** – (Chromosome).
- **parent2** – (Chromosome).

Returns
child1 and child2 (as Chromosomes).

Module contents

pygenalgo.operators.migration package

Submodules

pygenalgo.operators.migration.clockwise_migration module

class pygenalgo.operators.migration.clockwise_migration.**ClockwiseMigration**(*migration_probability: float = 0.95*)

Bases: *MigrationOperator*

Description:

Clockwise Migration implements a "very basic" migration policy in which each island migrates its best chromosome to the population on its right, following a "clockwise" rotation movement.

migrate(*islands: list[*SubPopulation*]*) → None

Perform the migration operation on the list of SubPopulations.

Parameters
islands – list[SubPopulation].

Returns
None.

pygenalgo.operators.migration.meta_migration module

class pygenalgo.operators.migration.meta_migration.**MetaMigration**(*migration_probability: float = 0.95*)

Bases: *MigrationOperator*

Description:

Meta-migrator, performs the migration between the subpopulations by applying randomly all other migrators (one at a time), with equal probability.

NOTE: In the future the equal probabilities can be amended.

property all_counters: dict

Accessor (getter) of the application counter from all the internal migrators. This is mostly to verify that everything is working as expected.

Returns
a dictionary with the counter calls for all migrator methods.

migrate(*islands: list[SubPopulation]*) → None

Perform the migration operation on the list of SubPopulations.

Parameters
islands – list[SubPopulation].

Returns
None.

reset_counter() → None

Sets ALL the counters to 'zero'. We have to override the super().reset_counter() method, because we have to call explicitly the reset_counter on all the internal operators.

Returns
None.

pygenalgo.operators.migration.migration_operator module

class pygenalgo.operators.migration.migration_operator.**MigrationOperator**(*migration_probability: float*)

Bases: *GeneticOperator*

Description:

Provides the base class (interface) for a Migration Operator.

migrate(*islands: list[SubPopulation]*)

Abstract method that "reminds" the user that if they want to create a Migration Class that inherits from here they should implement a migrate method.

Parameters
islands – list[SubPopulation].

Returns
Nothing but raising an error.

pygenalgo.operators.migration.random_migration module

class pygenalgo.operators.migration.random_migration.**RandomMigration**(*migration_probability: float = 0.95*)

Bases: *MigrationOperator*

Description:

Random Migration implements a "very basic" migration policy in which each island migrates its best chromosome to a randomly selected population.

migrate(*islands: list[*SubPopulation*]*) → None

Perform the migration operation on the list of SubPopulations.

Parameters
islands – list[SubPopulation].

Returns
None.

Module contents

pygenalgo.operators.mutation package

Submodules

pygenalgo.operators.mutation.flip_mutator module

class pygenalgo.operators.mutation.flip_mutator.**FlipMutator**(*mutate_probability: float = 0.1*)

Bases: *MutationOperator*

Description:

Flip mutator, mutates the chromosome by selecting randomly a position and flip its Gene value (0 -> 1, or 1 -> 0).

mutate(*individual:* Chromosome) → None

Perform the mutation operation by randomly flipping a gene.

Parameters
individual – (Chromosome).

Returns
None.

pygenalgo.operators.mutation.gaussian_mutator module

class pygenalgo.operators.mutation.gaussian_mutator.**GaussianMutator**(*mutate_probability: float = 0.1*)

Bases: *MutationOperator*

Description:

Gaussian mutator, mutates the chromosome by selecting randomly a position and add a Gaussian random value to the current gene value.

mutate(*individual:* Chromosome) → None

Perform the mutation operation by randomly adding the Gaussian value to a randomly selected gene position.

Parameters
individual – (Chromosome).

Returns
None.

pygenalgo.operators.mutation.inverse_mutator module

class pygenalgo.operators.mutation.inverse_mutator.**InverseMutator**(*mutate_probability: float = 0.1*)

Bases: *MutationOperator*

Description:

Inverse mutator mutates the chromosome by inverting the order of the gene values between two randomly selected gene end-positions.

mutate(*individual:* Chromosome) → None

Perform the mutation operation by inverting the genes between at two random positions.

Parameters
individual – (Chromosome).

Returns
None.

pygenalgo.operators.mutation.meta_mutator module

class pygenalgo.operators.mutation.meta_mutator.**MetaMutator**(*mutate_probability: float = 0.1*)

Bases: *MutationOperator*

Description:

Meta-mutator, mutates the chromosome by applying randomly all other mutators (one at a time), with equal probability.

NOTE: In the future the equal probabilities can be amended.

property all_counters: dict

Accessor (getter) of the application counter from all the internal mutators. This is mostly to verify that everything is working as expected.

Returns
a dictionary with the counter calls for all mutator methods.

mutate(*individual:* Chromosome) → None

Perform the mutation operation by randomly applying another mutator.

Parameters
individual – (Chromosome).

Returns
None.

reset_counter() → None

Sets ALL the counters to 'zero'. We have to override the super().reset_counter() method, because we have to call explicitly the reset_counter on all the internal operators.

Returns
None.

pygenalgo.operators.mutation.mutate_operator module

class pygenalgo.operators.mutation.mutate_operator.**MutationOperator**(*mutation_probability: float*)

Bases: *GeneticOperator*

Description:

Provides the base class (interface) for a Mutation Operator.

mutate(*individual:* Chromosome) → None

Abstract method that "reminds" the user that if they want to create a Mutation Class that inherits from here they should implement a mutate method.

Parameters
individual – the chromosome to be mutated.

Returns
Nothing but raising an error.

pygenalgo.operators.mutation.random_mutator module

class pygenalgo.operators.mutation.random_mutator.**RandomMutator**(*mutate_probability: float = 0.1*)

Bases: *MutationOperator*

Description:

Random mutator, mutates the chromosome by selecting randomly a position and replace the Gene with a new one that has been generated randomly (uniform probability).

mutate(*individual:* Chromosome) → None

Perform the mutation operation by randomly replacing a gene with a new one that has been generated randomly.

Parameters
individual – (Chromosome).

Returns
None.

pygenalgo.operators.mutation.shuffle_mutator module

class pygenalgo.operators.mutation.shuffle_mutator.**ShuffleMutator**(*mutate_probability: float = 0.1*)

Bases: *MutationOperator*

Description:

Shuffle mutator mutates the chromosome by shuffling the gene values between two randomly selected gene end-positions.

mutate(*individual:* Chromosome) → None

Perform the mutation operation by shuffling the genes between at two random positions.

Parameters
individual – (Chromosome).

Returns
None.

pygenalgo.operators.mutation.swap_mutator module

class pygenalgo.operators.mutation.swap_mutator.**SwapMutator**(*mutate_probability: float = 0.1*)

Bases: *MutationOperator*

Description:

Swap mutator mutates the chromosome by swapping the gene values between two randomly selected gene positions.

mutate(*individual:* Chromosome) → None

Perform the mutation operation by swapping the genes at two random positions.

Parameters
individual – (Chromosome).

Returns
None.

Module contents

pygenalgo.operators.selection package

Submodules

pygenalgo.operators.selection.boltzmann_selector module

class pygenalgo.operators.selection.boltzmann_selector.**BoltzmannSelector**(*select_probability: float = 1.0*, *k: float = 100.0*)

Bases: *SelectionOperator*

Description:

Boltzmann Selector implements an object that performs selection by choosing an individual from a set of individuals by sampling solutions from a Boltzmann distribution depending on their fitness's.

`select`(*population: list[*Chromosome*]*)

Select the individuals, from the input population, that will be passed on to the next genetic operations of crossover and mutation to form the new population of solutions.

NOTE: the Boltzmann constant is held in the '_items' variable.

Parameters
population – a list of chromosomes to select the parents from.

Returns
the selected parents population (as list of chromosomes).

pygenalgo.operators.selection.linear_rank_selector module

`class` pygenalgo.operators.selection.linear_rank_selector.`LinearRankSelector`(*select_probability: float = 1.0*)

Bases: *SelectionOperator*

Description:

Linear Rank Selector implements an object that performs selection using ranking. The individuals first are sorted according to their fitness values. The rank N is assigned to the best individual and the rank 1 to the worst individual.

After that the selection process is similar to the one of RouletteWheelSelector.

`select`(*population: list[*Chromosome*]*)

Select the individuals, from the input population, that will be passed on to the next genetic operations of crossover and mutation to form the new population of solutions.

Parameters
population – a list of chromosomes to select the parents from.

Returns
the selected parents population (as list of chromosomes).

pygenalgo.operators.selection.random_selector module

`class` pygenalgo.operators.selection.random_selector.`RandomSelector`(*select_probability: float = 1.0*)

Bases: *SelectionOperator*

Description:

Random Selector implements selection assuming that all members of the population have the same probability to be selected as parents 1/N, effectively assuming a uniform probability.

It does not favour the fit individuals therefore the mixing will be very slow.

`select`(*population: list[*Chromosome*]*)

Select the individuals, from the input population, that will be passed on to the next genetic operations of crossover and mutation to form the new population of solutions.

Parameters
population – a list of chromosomes to select the parents from.

Returns
the selected parents population (as list of chromosomes).

pygenalgo.operators.selection.roulette_wheel_selector module

class pygenalgo.operators.selection.roulette_wheel_selector.**RouletteWheelSelector**(*select_probability: float = 1.0*)

Bases: *SelectionOperator*

Description:

Roulette Wheel Selector implements 'fitness proportional selection'. Each member of the population is assigned a probability value that is directly proportional to its fitness value (compared to the rest of the population).

Individuals with higher fitness value are more likely to be selected for parents when forming the new generation of individuals (offsprings).

select(*population: list[*Chromosome*]*)

Select the individuals, from the input population, that will be passed on to the next genetic operations of crossover and mutation to form the new population of solutions.

Parameters
population – a list of chromosomes to select the parents from.

Returns
the selected parents population (as list of chromosomes).

pygenalgo.operators.selection.select_operator module

class pygenalgo.operators.selection.select_operator.**SelectionOperator**(*selection_probability: float*)

Bases: *GeneticOperator*

Description:

Provides the base class (interface) for a Selection Operator. Note that even though the operator accepts a probability value, for the moment this operator is applied with 100% probability.

select(*population: list[*Chromosome*]*)

Abstract method that "reminds" the user that if they want to create a Selection Class that inherits from here they should implement a select method.

Parameters
population – is a list, with the chromosomes, to select

the parents for the next generation

Returns
Nothing but raising an error.

pygenalgo.operators.selection.stochastic_universal_selector module

class pygenalgo.operators.selection.stochastic_universal_selector.**StochasticUniversalSelector**(*select_probability: float = 1.0*)

Bases: *SelectionOperator*

Description:

Stochastic Universal Selector is an extension of fitness proportionate selection (i.e. RouletteWheelSelection) which exhibits no bias and minimal spread. Where RWS chooses several solutions from the population by repeated random sampling, SUS uses a single random value to sample all the solutions by choosing them at evenly spaced intervals. This gives weaker members of the population (according to their fitness) a chance to be chosen.

select(*population: list[*Chromosome*]*)

Select the individuals, from the input population, that will be passed on to the next genetic operations of crossover and mutation to form the new population of solutions.

Parameters
population – a list of chromosomes to select the parents from.

Returns
the selected parents population (as list of chromosomes).

pygenalgo.operators.selection.tournament_selector module

class pygenalgo.operators.selection.tournament_selector.**TournamentSelector**(*select_probability: float = 1.0*, *k: int = 5*)

Bases: *SelectionOperator*

Description:

Tournament Selector implements an object that performs selection by choosing an individual from a set of individuals. The winner of each tournament i.e. (the one with the highest fitness value) is selected as new parent to perform crossover and mutation.

select(*population: list[*Chromosome*]*)

Select the individuals, from the input population, that will be passed on to the next genetic operations of crossover and mutation to form the new population of solutions.

Parameters
population – a list of chromosomes to select the parents from.

Returns
the selected parents population (as list of chromosomes).

pygenalgo.operators.selection.truncation_selector module

class pygenalgo.operators.selection.truncation_selector.**TruncationSelector**(*select_probability: float = 1.0*, *p: float = 0.3*)

Bases: *SelectionOperator*

Description:

Truncation Selector, creates a new population using a pre-defined proportion of the old population. When this method is called, it sorts the individuals of the OLD population using their fitness and then using a predefined value (e.g. p=0.3 or 30%) selects repeatedly new individuals from the top 0.3 percent of the old population, until we reach the required size of the NEW population.

select(*population: list[*Chromosome*]*)

Select the individuals, from the input population, that will be passed on to the next genetic operations of crossover and mutation to form the new population of solutions.

Parameters
population – a list of chromosomes to select the parents from.

Returns
the selected parents population (as list of chromosomes).

Module contents

Submodules

pygenalgo.operators.genetic_operator module

class pygenalgo.operators.genetic_operator.**GeneticOperator**(*_probability: float*)

Bases: object

Description:

Provides the base class (interface) for a Genetic Operator. This class includes some common variables (such as the probability and the application counter) along with access to them.

All genetic operators (Selection, Crossover, Mutation, Migration) should inherit this class.

property counter: int

Accessor (getter) of the application counter.

Returns
the int value of the counter variable.

inc_counter() → None

Increase the counter value by one. This is applied after each application of the genetic operator.

Returns
None.

property items: list | tuple

Accessor (getter) of the _items container.

Returns
_items (if any).

property iter: int

Accessor (getter) of the iteration parameter.

Returns
the iteration value.

property probability: float

Accessor (getter) of the probability.

Returns
the float value of the probability.

reset_counter() → None

Sets the counter value to zero.

Returns
None.

property rng

Get access of the Class variable (_rng).

Returns
the random number generator.

Module contents

1.1.2 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pygenalgo, 22
pygenalgo.engines, 6
pygenalgo.engines.auxiliary, 1
pygenalgo.engines.generic_ga, 2
pygenalgo.engines.island_model_ga, 3
pygenalgo.engines.standard_ga, 5
pygenalgo.genome, 8
pygenalgo.genome.chromosome, 6
pygenalgo.genome.gene, 7
pygenalgo.operators, 22
pygenalgo.operators.crossover, 12
pygenalgo.operators.crossover.crossover_operator, 8
pygenalgo.operators.crossover.meta_crossover, 9
pygenalgo.operators.crossover.mutli_point_crossover, 9
pygenalgo.operators.crossover.order_crossover, 10
pygenalgo.operators.crossover.partially_mapped_crossover, 10
pygenalgo.operators.crossover.position_based_crossover, 11
pygenalgo.operators.crossover.single_point_crossover, 11
pygenalgo.operators.crossover.uniform_crossover, 12
pygenalgo.operators.genetic_operator, 21
pygenalgo.operators.migration, 14
pygenalgo.operators.migration.clockwise_migration, 12
pygenalgo.operators.migration.meta_migration, 13
pygenalgo.operators.migration.migration_operator, 13
pygenalgo.operators.migration.random_migration, 14
pygenalgo.operators.mutation, 17
pygenalgo.operators.mutation.flip_mutator, 14
pygenalgo.operators.mutation.gaussian_mutator, 14
pygenalgo.operators.mutation.inverse_mutator, 15
pygenalgo.operators.mutation.meta_mutator, 15
pygenalgo.operators.mutation.mutate_operator, 16
pygenalgo.operators.mutation.random_mutator, 16
pygenalgo.operators.mutation.shuffle_mutator, 17
pygenalgo.operators.mutation.swap_mutator, 17
pygenalgo.operators.selection, 21
pygenalgo.operators.selection.boltzmann_selector, 17
pygenalgo.operators.selection.linear_rank_selector, 18
pygenalgo.operators.selection.random_selector, 18
pygenalgo.operators.selection.roulette_wheel_selector, 19
pygenalgo.operators.selection.select_operator, 19
pygenalgo.operators.selection.stochastic_universal_selecto
20
pygenalgo.operators.selection.tournament_selector, 20
pygenalgo.operators.selection.truncation_selector, 21

INDEX

A

all_counters (*pygenalgo.operators.crossover.meta_crossover.MetaCrossover property*), 9
all_counters (*pygenalgo.operators.migration.meta_migration.MetaMigration property*), 13
all_counters (*pygenalgo.operators.mutation.meta_mutator.MetaMutator property*), 15
apply_corrections() (*in module pygenalgo.engines.auxiliary*), 1
avg_hamming_dist() (*in module pygenalgo.engines.auxiliary*), 1

B

best_chromosome() (*pygenalgo.engines.generic_ga.GenericGA method*), 2
BoltzmannSelector (*class in pygenalgo.operators.selection.boltzmann_selector*), 17

C

Chromosome (*class in pygenalgo.genome.chromosome*), 6
ClockwiseMigration (*class in pygenalgo.operators.migration.clockwise_migration*), 12
clone() (*pygenalgo.genome.chromosome.Chromosome method*), 6
clone() (*pygenalgo.genome.gene.Gene method*), 7
counter (*pygenalgo.operators.genetic_operator.GeneticOperator property*), 21
crossover() (*pygenalgo.operators.crossover.crossover_operator.CrossoverOperator method*), 8
crossover() (*pygenalgo.operators.crossover.meta_crossover.MetaCrossover method*), 9
crossover() (*pygenalgo.operators.crossover.mutli_point_crossover.MultiPointCrossover method*), 9
crossover() (*pygenalgo.operators.crossover.order_crossover.OrderCrossover method*), 10
crossover() (*pygenalgo.operators.crossover.partially_mapped_crossover.PartiallyMappedCrossover method*), 10
crossover() (*pygenalgo.operators.crossover.position_based_crossover.PositionBasedCrossover method*), 11
crossover() (*pygenalgo.operators.crossover.single_point_crossover.Single method*), 11
crossover() (*pygenalgo.operators.crossover.uniform_crossover.UniformC method*), 12
crossover_mutate() (*pygenalgo.engines.generic_ga.GenericGA method*), 2
crossover_op (*pygenalgo.engines.generic_ga.GenericGA property*), 2
CrossoverOperator (*class in pygenalgo.operators.crossover.crossover_operator*), 8

E

evaluate_fitness() (*pygenalgo.engines.generic_ga.GenericGA method*), 2
evaluate_fitness() (*pygenalgo.engines.island_model_ga.IslandModelGA method*), 3
evaluate_fitness() (*pygenalgo.engines.standard_ga.StandardGA method*), 5
evolve_population() (*pygenalgo.engines.island_model_ga.IslandModelGA class method*), 4

F

fitness (*pygenalgo.genome.chromosome.Chromosome property*), 6
fitness_func (*pygenalgo.engines.generic_ga.GenericGA attribute*), 2
fitness_func (*pygenalgo.engines.standard_ga.StandardGA attribute*), 5
flip() (*pygenalgo.genome.gene.Gene method*), 7
FlipMutator (*class in pygenalgo.operators.mutation.flip_mutator*), 14

G

gaussian() (*pygenalgo.genome.gene.Gene method*), 7

GaussianMutator (*class in pygenalgo.operators.mutation.gaussian_mutator*), 14
Gene (*class in pygenalgo.genome.gene*), 7
GenericGA (*class in pygenalgo.engines.generic_ga*), 2
GeneticOperator (*class in pygenalgo.operators.genetic_operator*), 21
genome (*pygenalgo.genome.chromosome.Chromosome property*), 6

H

hamming_distance() (*pygenalgo.genome.chromosome.Chromosome method*), 7

I

id (*pygenalgo.engines.auxiliary.SubPopulation property*), 1
inc_counter() (*pygenalgo.operators.genetic_operator.GeneticOperator method*), 21
individual_fitness() (*pygenalgo.engines.generic_ga.GenericGA method*), 2
InverseMutator (*class in pygenalgo.operators.mutation.inverse_mutator*), 15
is_genome_valid() (*pygenalgo.genome.chromosome.Chromosome method*), 7
is_valid (*pygenalgo.genome.gene.Gene property*), 8
IslandModelGA (*class in pygenalgo.engines.island_model_ga*), 3
items (*pygenalgo.operators.genetic_operator.GeneticOperator property*), 21
iter (*pygenalgo.operators.genetic_operator.GeneticOperator property*), 21

L

LinearRankSelector (*class in pygenalgo.operators.selection.linear_rank_selector*), 18

M

MAX_CPUs (*pygenalgo.engines.generic_ga.GenericGA attribute*), 2
MetaCrossover (*class in pygenalgo.operators.crossover.meta_crossover*), 9
MetaMigration (*class in pygenalgo.operators.migration.meta_migration*), 13
MetaMutator (*class in pygenalgo.operators.mutation.meta_mutator*), 15
migrate() (*pygenalgo.operators.migration.clockwise_migration.Clockwise method*), 12
migrate() (*pygenalgo.operators.migration.meta_migration.MetaMigration method*), 13
migrate() (*pygenalgo.operators.migration.migration_operator.MigrationO method*), 13
migrate() (*pygenalgo.operators.migration.random_migration.RandomMig method*), 14
migrate_op (*pygenalgo.engines.island_model_ga.IslandModelGA property*), 4
MigrationOperator (*class in pygenalgo.operators.migration.migration_operator*), 13
module
    pygenalgo, 22
    pygenalgo.engines, 6
    pygenalgo.engines.auxiliary, 1
    pygenalgo.engines.generic_ga, 2
    pygenalgo.engines.island_model_ga, 3
    pygenalgo.engines.standard_ga, 5
    pygenalgo.genome, 8
    pygenalgo.genome.chromosome, 6
    pygenalgo.genome.gene, 7
    pygenalgo.operators, 22
    pygenalgo.operators.crossover, 12
    pygenalgo.operators.crossover.crossover_operator, 8
    pygenalgo.operators.crossover.meta_crossover, 9
    pygenalgo.operators.crossover.mutli_point_crossover, 9
    pygenalgo.operators.crossover.order_crossover, 10
    pygenalgo.operators.crossover.partially_mapped_crossov 10
    pygenalgo.operators.crossover.position_based_crossover 11
    pygenalgo.operators.crossover.single_point_crossover, 11
    pygenalgo.operators.crossover.uniform_crossover, 12
    pygenalgo.operators.genetic_operator, 21
    pygenalgo.operators.migration, 14
    pygenalgo.operators.migration.clockwise_migration, 12
    pygenalgo.operators.migration.meta_migration, 13
    pygenalgo.operators.migration.migration_operator, 13
    pygenalgo.operators.migration.random_migration, 14

pygenalgo.operators.mutation, 17
pygenalgo.operators.mutation.flip_mutator, 14
pygenalgo.operators.mutation.gaussian_mutator, 14
pygenalgo.operators.mutation.inverse_mutator, 15
pygenalgo.operators.mutation.meta_mutator, 15
pygenalgo.operators.mutation.mutate_operator, 16
pygenalgo.operators.mutation.random_mutator, 16
pygenalgo.operators.mutation.shuffle_mutator, 17
pygenalgo.operators.mutation.swap_mutator, 17
pygenalgo.operators.selection, 21
pygenalgo.operators.selection.boltzmann_selector, 17
pygenalgo.operators.selection.linear_rank_selector, 18
pygenalgo.operators.selection.random_selector, 18
pygenalgo.operators.selection.roulette_wheel_selector, 19
pygenalgo.operators.selection.select_operator, 19
pygenalgo.operators.selection.stochastic_universal_selector, 20
pygenalgo.operators.selection.tournament_selector, 20
pygenalgo.operators.selection.truncation_selector, 21
MultiPointCrossover (*class in pygenalgo.operators.crossover.mutli_point_crossover*), 9
mutate() (*pygenalgo.operators.mutation.flip_mutator.FlipMutator method*), 14
mutate() (*pygenalgo.operators.mutation.gaussian_mutator.GaussianMutator method*), 14
mutate() (*pygenalgo.operators.mutation.inverse_mutator.InverseMutator method*), 15
mutate() (*pygenalgo.operators.mutation.meta_mutator.MetaMutator method*), 15
mutate() (*pygenalgo.operators.mutation.mutate_operator.MutationOperator method*), 16
mutate() (*pygenalgo.operators.mutation.random_mutator.RandomMutator method*), 16
mutate() (*pygenalgo.operators.mutation.shuffle_mutator.ShuffleMutator method*), 17
mutate() (*pygenalgo.operators.mutation.swap_mutator.SwapMutator method*), 17
mutate_op (*pygenalgo.engines.generic_ga.GenericGA property*), 3
MutationOperator (*class in pygenalgo.operators.mutation.mutate_operator*), 16

N

num_islands (*pygenalgo.engines.island_model_ga.IslandModelGA attribute*), 4

O

OrderCrossover (*class in pygenalgo.operators.crossover.order_crossover*), 10

P

PartiallyMappedCrossover (*class in pygenalgo.operators.crossover.partially_mapped_crossover*), 10
pop_id (*pygenalgo.engines.auxiliary.SubPopulation attribute*), 1
population (*pygenalgo.engines.auxiliary.SubPopulation attribute*), 1
population (*pygenalgo.engines.generic_ga.GenericGA attribute*), 3
population (*pygenalgo.engines.standard_ga.StandardGA attribute*), 5
population_fitness() (*pygenalgo.engines.generic_ga.GenericGA method*), 3
PositionBasedCrossover (*class in pygenalgo.operators.crossover.position_based_crossover*), 11
print_migration_stats() (*pygenalgo.engines.island_model_ga.IslandModelGA method*), 4
print_operator_stats() (*pygenalgo.engines.standard_ga.StandardGA method*), 5
probability (*pygenalgo.operators.genetic_operator.GeneticOperator property*), 22
pygenalgo module, 22
pygenalgo.engines module, 6
pygenalgo.engines.auxiliary module, 1
pygenalgo.engines.generic_ga module, 2
pygenalgo.engines.island_model_ga module, 3
pygenalgo.engines.standard_ga module, 5
pygenalgo.genome module, 8

pygenalgo.genome.chromosome
    module, 6
pygenalgo.genome.gene
    module, 7
pygenalgo.operators
    module, 22
pygenalgo.operators.crossover
    module, 12
pygenalgo.operators.crossover.crossover_operator
    module, 8
pygenalgo.operators.crossover.meta_crossover
    module, 9
pygenalgo.operators.crossover.mutli_point_crossover
    module, 9
pygenalgo.operators.crossover.order_crossover
    module, 10
pygenalgo.operators.crossover.partially_mapped_crossover
    module, 10
pygenalgo.operators.crossover.position_based_crossover
    module, 11
pygenalgo.operators.crossover.single_point_crossover
    module, 11
pygenalgo.operators.crossover.uniform_crossover
    module, 12
pygenalgo.operators.genetic_operator
    module, 21
pygenalgo.operators.migration
    module, 14
pygenalgo.operators.migration.clockwise_migration
    module, 12
pygenalgo.operators.migration.meta_migration
    module, 13
pygenalgo.operators.migration.migration_operator
    module, 13
pygenalgo.operators.migration.random_migration
    module, 14
pygenalgo.operators.mutation
    module, 17
pygenalgo.operators.mutation.flip_mutator
    module, 14
pygenalgo.operators.mutation.gaussian_mutator
    module, 14
pygenalgo.operators.mutation.inverse_mutator
    module, 15
pygenalgo.operators.mutation.meta_mutator
    module, 15
pygenalgo.operators.mutation.mutate_operator
    module, 16
pygenalgo.operators.mutation.random_mutator
    module, 16
pygenalgo.operators.mutation.shuffle_mutator
    module, 17
pygenalgo.operators.mutation.swap_mutator
    module, 17
pygenalgo.operators.selection
    module, 21
pygenalgo.operators.selection.boltzmann_selector
    module, 17
pygenalgo.operators.selection.linear_rank_selector
    module, 18
pygenalgo.operators.selection.random_selector
    module, 18
pygenalgo.operators.selection.roulette_wheel_selector
    module, 19
pygenalgo.operators.selection.select_operator
    module, 19
pygenalgo.operators.selection.stochastic_universal_selecto
    module, 20
pygenalgo.operators.selection.tournament_selector
    module, 20
pygenalgo.operators.selection.truncation_selector
    module, 21

R

Random() (*pygenalgo.genome.gene.Gene method*), 8
RandomMigration (*class in pygenalgo.operators.migration.random_migration*), 14
RandomMutator (*class in pygenalgo.operators.mutation.random_mutator*), 16
RandomSelector (*class in pygenalgo.operators.selection.random_selector*), 18
reset_counter() (*pygenalgo.operators.crossover.meta_crossover.MetaCrossover method*), 9
reset_counter() (*pygenalgo.operators.genetic_operator.GeneticOperator method*), 22
reset_counter() (*pygenalgo.operators.migration.meta_migration.MetaMigration method*), 13
reset_counter() (*pygenalgo.operators.mutation.meta_mutator.MetaMutator method*), 15
rng (*pygenalgo.operators.genetic_operator.GeneticOperator property*), 22
rng_GA (*pygenalgo.engines.generic_ga.GenericGA attribute*), 3
RouletteWheelSelector (*class in pygenalgo.operators.selection.roulette_wheel_selector*), 19
run() (*pygenalgo.engines.generic_ga.GenericGA method*), 3
run() (*pygenalgo.engines.island_model_ga.IslandModelGA method*), 4

run() (*pygenalgo.engines.standard_ga.StandardGA method*), 5

S

select() (*pygenalgo.operators.selection.boltzmann_selector.BoltzmannSelector method*), 18
select() (*pygenalgo.operators.selection.linear_rank_selector.LinearRankSelector method*), 18
select() (*pygenalgo.operators.selection.random_selector.RandomSelector method*), 18
select() (*pygenalgo.operators.selection.roulette_wheel_selector.RouletteWheelSelector method*), 19
select() (*pygenalgo.operators.selection.select_operator.SelectionOperator method*), 19
select() (*pygenalgo.operators.selection.stochastic_universal_selector.StochasticUniversalSelector method*), 20
select() (*pygenalgo.operators.selection.tournament_selector.TournamentSelector method*), 20
select() (*pygenalgo.operators.selection.truncation_selector.TruncationSelector method*), 21
select_op (*pygenalgo.engines.generic_ga.GenericGA property*), 3
SelectionOperator (*class in pygenalgo.operators.selection.select_operator*), 19
ShuffleMutator (*class in pygenalgo.operators.mutation.shuffle_mutator*), 17
SinglePointCrossover (*class in pygenalgo.operators.crossover.single_point_crossover*), 11
StandardGA (*class in pygenalgo.engines.standard_ga*), 5
stats (*pygenalgo.engines.generic_ga.GenericGA property*), 3
StochasticUniversalSelector (*class in pygenalgo.operators.selection.stochastic_universal_selector*), 20
SubPopulation (*class in pygenalgo.engines.auxiliary*), 1
SwapMutator (*class in pygenalgo.operators.mutation.swap_mutator*), 17

T

TournamentSelector (*class in pygenalgo.operators.selection.tournament_selector*), 20
TruncationSelector (*class in pygenalgo.operators.selection.truncation_selector*), 21

U

UniformCrossover (*class in pygenalgo.operators.crossover.uniform_crossover*), 12
update_stats() (*pygenalgo.engines.standard_ga.StandardGA method*), 6

V

validate() (*pygenalgo.genome.chromosome.Chromosome property*), 7
validate() (*pygenalgo.genome.gene.Gene property*), 8