Genetic Documentation

Release 1.1

Ashwin Panchapakesan

February 25, 2017

Contents

Contents:

CHAPTER 1

Required Installations

Python 2.7

This package requires python 2.7 and will not work with python 2.6. It can easily be transformed into python 2.6 compliant code. Most of the incompatibilities come from list/dict comprehension expressions

PyGame (and all its dependencies)

Used in visualization of the included TSP solver

pycontract

Used for contract checking (only when testmode is True; see the documentation for settings.py)

CHAPTER 2

Overview - How to use this Framework

If you are making your own evolutionary algorithm, there are several files that you should edit

individual.py

This file defines what an individual is (in a class). Generically, an `individual` is simply an ordered collection of chromosomes. The original implementation treats each chromosome differently. Therefore, all the chromosomes of an individual are maintained in a `list` as opposed to a `set`.

Also implemented in this file are methods to the `individual` class that help identify an individual, define its `hash`, test its equality to another `individual` instance, etc

population.py

This file contains the functions that define population generation. The important function defined here is *genPop()*, which may be used as an interface to creating unique individuals.

genPop(*N*, *chromGenfuncs*, *chromGenParams*)

Parameters

- **N** (*int*) – the number of individuals in the population
- **chromGenfuncs** (*list of tuples*) – a list of functions. The ith function is responsible for generating the ith chromosome for the individual. The length of this list is exactly the number of chromosomes in each individual
- **chromGenfuncs** – list of params for the functions in chromGenfuncs. The ith function in chromGenfuncs is called with the parameters held in the ith tuple of this list

Return type list of unique individuals. Uniqueness is defined by `Individual.__eq__`

chromGenfuncs

chromGenfuncs is a list of functions. The idea here is that each individual in the population is made up of C chromosomes. These C chromosomes are generated independently of each other for each individual in the initial population. Therefore, there must be exactly C functions listed in chromGenfuncs. The `i` th function in chromGenfuncs will be used to generate the `i` th chromosome of each individual

chromGenParams

chromGenParams is a list of tuples. There should be exactly as many tuples in this list, as there are functions in chromGenfuncs. To generate the ith chromosome for each individual in the population, the ith function in chromGenfuncs is called with the parameters in the ith tuple of chromGenParams as follows:

```
chromGenfuncs[i](*chromGenParams[i])
```

Though that is the general idea behind how *genPop()* works, it actually performs this call in a for loop over a `zip()` of chromGenfuncs and chromGenParams

---

Note: In order for *genPop()* to work, `Individual` must implement `__hash__()`. This is because *genPop()* uses a `set` internally before returning a list of individuals as the generated population. As a result, a meaningful `__hash__()` must be implemented in `Individual`.

---

fitness.py

This file contains all selection functions for evaluating the fitness of any individual of the population. The main function in this file is called *score()*.

score(*p*, *scorefuncs*, *scorefuncparams*, *SCORES*)

Parameters

- **p** (*instance of individual*) – the individual being evaluated
- **scorefuncs** (*list of functions*) – a list of functions - one to evaluate each chromosome in the individual
- **scorefuncparams** – a list of tuples containing the parameters for each of the functions in scorefuncs. Each function will be called by calling scorefuncs[i](p, *scorefuncparams[i])
- **SCORES** (*dict {Individual : number}*) – a dict mapping instances of Individual to their fitness

Return type number (the fitness of the individual)

Note: if the individual being evaluated by this function (`p`) was not in `SCORES` before the function is executed, it will be inserted into `SCORES` by this function. Thus, `SCORES` is modified in-place by this function as required.

---

selection.py

This file contains all selection functions for selecting individuals from a population for any purpose.

There are three important functions already implemented:

1. *tournamentSelect()*
2. *rouletteWheelSelect()*
3. *getRouletteWheel()*

tournamentSelect()

tournamentSelect(*pop*, *T*, *w*, *n*, *scorefunc*, *scoreparams*)

Parameters

- **pop** (*list of Individuals*) – the population to select from
- **T** (*int*) – the number of contestants in each tournament (must be smaller than len(pop))
- **w** (*int*) – the number of winners in each tournament (must be smaller than T)
- **n** (*int*) – the number of Individuals to be selected from the population by Tournament Selection (n%w should be 0)
- **scorefunc** (*function*) – the function used to evaluate the fitness of individuals, to determine the winner(s) of a tournament
- **scoreparams** (*tuple*) – the parameters that scorefunc requires, other than the individual itself. The individual is provided along with the unpacked list of params

Return type list of individuals

rouletteWheelSelect()

rouletteWheelSelect(*wheel*, *s=None*)

Parameters

- **wheel** (*a list of 3-tuples. Each tuple consists of the individual, the lower bound (float) of its section of the roulette wheel, and the upper bound (float) of its section of the roulette wheel.*) – a roulette wheel
- **s** (*float*) – the random number on which the roulette ball lands on the roulette wheel. This is not provided when calling the function (though it may be if desired). Using this, a binary search is performed to find the Individual that bet on a section of the roulette wheel containing this number

Return type single individual

getRouletteWheel()

getRouletteWheel(*pop*, *SCORES*)

Parameters

- **pop** (*list of instances of Individual*) – the population for which a roulette wheel must be made
- **SCORES** (*dict {Individual:number}*) – a dictionary that maps instances of Individual to their fitnesses

Return type list of 3-tuples (Individual, lowerBound, UpperBound)

crossover.py

All functions that have to do with crossing over chromosomes between individuals are defined here. There is no generic driver function here as crossovers are defined per GA.

mutation.py

All functions that have to do with mutating chromosomes between individuals are defined here. There is no generic driver function here as mutations are defined per GA

CHAPTER 3

individual.py

Define an individual to be used for evolution.

Class Variables

ID

A trackable ID generator

Instance variables

id

A trackable identifier for the individual

chromosomes

An ordered collection of the genetic material of this individual. Maintained as a list

Methods

__eq__(self, other)

Return True if all chromosomes of self and other are equal (and in the same order). Else, return False

__hash__(self)

Return the hash of the tuple version of all chromosomes

`__len__(self)` Return the number of chromosomes `self` is made of.

`__getitem__(self, i)`

Return the *i* th individual

`__setitem__(self, index, obj)`

Set obj as the *index'th chromosome of ''self'*

`__contains__(self, chromosome)`

Return True if `chromosome` is a member of `self.chromosomes`. Else return False

`__repr__(self)`

Return `self.id` as a string

`append(self, chrom)`

Append `chrom` to `self.chromosomes`

`count(self, sub, chrom)` Return the number of occurrences of `sub` in the `chrom` th chromosome of `self`

CHAPTER 4

population.py

genPop(N, chromGenfuncs, chromGenParams)

Return a population (list) of N unique individuals. Each individual has len(chromgGenFuncs) chromosomes. For each individual, chromosome_i is generated by calling chromGenFuncs_i(chromeGenParams_i)

genCharsChrom(l, chars)

Return a chromosome (list) of length `l`, each of which is made up of the characters from chars.

genTour(numCities)

This is the chromosome generation function for the traveling salesman problem. This function returns a list of ints. This list is a permeation of {0, 1, 2, . . . , numCities-1} and represents a tour that the traveling salesman would take

CHAPTER 5

fitness.py

score(p, scorefuncs, scorefuncparams, SCORES)

Return the sum of the fitness of each chromosome of individual p and store the result in SCORES (mapped under p)
The score of the chromosome_i is determined by the call scorefunc[i](p.chromosomes[i], *scorefuncparams[i])

scoreOnes(p)

Return the number of '1's in the chromosome p

scoreTSP(tour, DIST)

Return the total distance of `tour`, a list of ints, representing a tour (each int is a city ID). `DIST` is a dictionary: {source_city_id : {destination_city_id : distance between source_city and desitantion_city} }

CHAPTER 6

selection.py

getRouletteWheel(pop, SCORES)

Return a fitness proportional roulette wheel. A roulette wheel is a list of 3-tuples structured as follows: (indiv, low, high) where indiv is the individual that bets on the section of the roulette wheel between low and high

rouletteWheelSelect(wheel, s=None)

Perform roulette wheel selection. A wheel is a fitness proportional roulette wheel as returned by the makeRoulette-Wheel function. The parameter s is not required thought not disallowed at the time of calling by the evolutionary algorithm. If it is not supplied, it will be set as a random float between 0 and 1. This function returns the individual that bet on the section of the roulette wheel that contains s

tournamentSelect(pop, T, w, n, scorefunc, scoreparams)

Return a list of n indivuduals. Each of these individuals has been selected by conducting tournaments of size T. Each tournament may have exactly w winners Winners of the tournament are the fittest individuals in the tournament as determined by scorefunc

CHAPTER 7

crossover.py

crossOnes(p1, p2)

Take two chromosomes (`p1` and `p2`). Cross them over. Return two new child chromosomes

injectionco(p1, p2, chrom)

Take two chromosomes `p1` and `p2`. Crossover them over as follows:

1. Select distinct points A < B between 0 and `len(p1.chromosomes[chrom])`
2. Make an empty child chromosome of length `len(p1.chromosomes[chrom])`
3. Copy over the genes of `p1` from A to (but not including) B into the corresponding genes of the child
4. Fill in the rest of the genes of the child with the genes from `p2`, in the order in which they appear in `p2`, making sure not to include alleles that already exist in the child

Return the child chromosome

twoChildCrossover(p1,p2, crossfuncs, crossparams)

Take two parents, `p1` and `p2`. Assume that `crossfuncs` and `crossparams` are of equal length Make two empty individuals to be returned as the answer For each `i``th pair of corresponding chromosomes in ``p1` and `p2`, cross them over with the corresponding `i``th function in ``crossfuncs` (and pass in the `i``th tuple of parameters from ``crossparams`) It is assumed that each function in `crossparams` returns two child chromosomes. When crossover on each pair of chromosomes is complete, add the first child chromosome to the first child individual to be returned and the second child chromosome to the second child individual to be returned. When all crossover operations are complete, return the two child individuals as the product of crossing over `p1` and `p2`.

oneChildCrossover(p1, p2, crossfuncs, crossparams)

Take two parents, `p1` and `p2`. Assume that `crossfuncs` and `crossparams` are of equal length Make an empty individual to be returned as the answer For each `i``th pair of corresponding chromosomes in ``p1` and `p2`, cross them over with the corresponding `i``th function in ``crossfuncs` (and pass in the `i``th tuple of parameters from ``crossparams`) It is assumed that each function in `crossparams` returns

one child chromosome When crossover on each pair of chromosomes is complete, add the child chromosome to the child individual to be returned When all crossover operations are complete, return the child individual as the product of crossing over `p1` and `p2`.

CHAPTER 8

mutaion.py

mutateSingleAllele(p, chrom, chars)

Return a new individual, which is the same as `p`, but with the `chrom` th chromosome changed as follows: Select a random gene and change its value to something from the choices in `chars`

swapmut(p, chrom)

Get the `chrom` th individual in p. Select two random elements in that chromosome and swap their positions in that chromosome Return a new individual that is the same as `p`, but with the above change made to its `chrom` th chromosome

revmut(p, chrom)

Get the `chrom` th individual in p. Select two random elements in that chromosome and reverse the order of genes between those two elements in that chromosome Return a new individual that is the same as `p`, but with the above change made to its `chrom` th chromosome

shufflemut(p, chrom)

Get the `chrom` th individual in p. Shuffle that chromosome with `random.shuffle` Return a new individual that is the same as `p`, but with the above change made to its `chrom` th chromosome

CHAPTER 9

GA.py

runTSPGA(kwargs)

Run a GA that solves the Traveling Salesman Problem with the settings generated in settings.py

runGA(kwargs)

Run a simple fitness-maximizing GA that solves any applicable problem. At the time of writing this document, it was applied to the One-Max problem

run(kwargs)

This is the main driver function that runs the required evolutionary algorithm. But before it does that, it also checks the sanity and makes sure to import PyContract if required (as indicated by the `testmode` flag)

CHAPTER 10

settings.py

Since the evolutionary framework requires several settings for each of the modules being used, a separate file is used to specify the correct settings for a run of an evolution.

This file contains functions, each of which generates a specific set of settings to run evolution on a specific problem

Fields

All fields have to be set with some value or other. There are no default values in this framework, by design

algorithm

The evolutionary algorithm to be run (defined in GA.py)

testmode

Set this to true to run in-function assert statements that check contracts. False otherwise

maxGens

The maximum number of generations for which evolution shall be run after which it will be stopped even if an optimal solution has not yet been discovered

targetscore

The known optimal fitness score of the problem. Setting this to `None` or `''` will simulate negative and positive infinity, respectively

popsize

The number of individuals in the population during evolution

numCrossovers

The number of crossover operations per generation

SCORES

A dictionary that remembers the fitness values of all individuals. This is used as an optimization. Usually, this is an empty dictionary This can be deactivated by changing `Individual.__hash__` to something that will be unique to each individual, regardless of genetic makeup

genfunc

The function that generates the initial population

genparams

A tuple containing the parameters to send to `genfunc` in the correct order

scorefunc

The function that returns the fitness evaluation of an individual. By default this is set to `fitness.score`. .. note:

```
It is advisable to leave this as ``fitness.score``, especially for multi-chromosome individuals.
```

scoreparams

This is a 3-tuple

Index	Type	Description
0	list of functions	the *i* th function listed here will be used to compute the fitness of the *i* th chromosome of the individuals
1	list of tuples	the *i* th tuple listed here contains the parameters (in the correct order) for the *i* th function in the list in index 0
2	dictionary	`SCORES`

Warning: The parameters listed do NOT include any reference to the individual whose fitness will be computed. The individual will be supplied by the main evolution function itself. This is because the individual is chosen by the selection function and therefore cannot be known at the time of making these settings

selectfunc

The selection function by which individuals will be selected for crossover

selectparams

A tuple of parameters (in the correct order) for the selection function

Warning: The parameters listed do NOT include any reference to the population from which individuals will be selected. The population will be supplied by the main evolution function itself. This is because the population keeps changing over time and therefore cannot be known at the time of making these settings

crossfunc

The function that performs crossover between two individuals. This is usually either `oneChildCrossover` or `twoChildCrossover`. These crossover functions return 1 or 2 children as the result of crossover, respectively.

crossfuncs

A list of crossover functions. The `i``th function in this list will be used (along with the ``i``th tuple of parameters from ``crossparams` to crossover the “i“th pair of corresponding chromosomes of two individuals.

crossparams

A tuple of parameters (in the correct order) for the crossover function

Warning: The parameters listed do NOT include any reference to the individuals to be crossed over. These individuals will be supplied by the main evolution function itself. This is because the individuals are chosen by the selection function and therefore cannot be known at the time of making these settings

mutfunc

The function that will mutate a given individual

mutparams

A tuple of parameters (in the correct order) for the crossover function

Warning: The parameters listed do NOT include any reference to the individual to be mutated. This individual will be supplied by the main evolution function itself. This is because the individual is chosen at random (with probability) and therefore cannot be known at the time of making these settings

crossprob

The probability of crossover occurring. Represented as a float in [0, 1]

mutprob

The probability of mutation occurring. Represented as a float in [0, 1]

rouletteWheelRequireres

A set of functions that require a roulette wheel. This is used later in the automated computation some settings

getWheel

A `bool` that determines whether the evolutionary algorithm must compute a roulette wheel for selection

Warning: This is an automatically set parameter. Do not alter it.

visualize

A boolean flag that determines if visualization is enabled

screenWidth

The width of the screen created for visualization

screenHeight

The height of the screen created for visualization

makeScreenParams

A tuple of parameters (in the correct order) required to make the screen on which the visualization will be drawn

drawParams

A tuple of parameters (in the correct order) required to draw the visualization on the screen

fon

The font with which any text should be written on screen during visualization

fontParams

The parameters for rendering font as a tuple 9in the correct order)

labelParams

A tuple of parameters (in the correct order) required to place any text in the correct place on screen during visualization

sanity

A list of parameter names that should be present in the settings. The settings are checked for the entries in `sanity` before any evolution is run, to ensure that all parameters are provided

answer

A dictionary of the settings to run evolution

Warning: It is generally a bad idea to alter statements that are not assignment statements. This is because they are automations that generate some settings, thus taking the responsibility of generating those settings away from the programmer. Altering them may have unintended side-effects

Warning: It is generally a bad idea to alter statements that are inside the `if visualize` block. This is a block that automates the inclusion of settings (both into the returned settings and the sanity) for visualization if it is enabled

CHAPTER 11

visualization.py

This module handles all the visualization for any evolution. The current implementation uses `pygame`

makeScreen(W, H)

Make an empty screen of width `W` and height `H`

normalize(point, (olow, ohigh), (low, high))

This is a helper function. It takes a value for `point`, originally measured in the scale [olow, ohigh]. The returned value is the corresponding value of `point` on the scale [low, high]

draw(tour, window, W, H, SCORES, COORDS)

Draws a tour of a traveling salesman, and writes the score of the tour on the window.

Each city in the tour is represented as a red dot, with white lines connecting them.

`COORDS` is a dictionary that contains the coordinates of the various cities.

The fitness score of `tour` is also written to `window`

killscreen()

This function cleans up pygame and destroys the window and the screen; to be called at the end of evolution.

CHAPTER 12

Contracts

Contracts are used to check the pre and post conditions of functions to make sure that the evolutionary algorithm remains constrained within the solution space.

All contracts used by all functions are listed here. It is highly recommended that similar functions that are implemented in the future implement similar contracts. This will be explained further as each contract is explaioned.

GA.py

The main GA driver has the following contracts. It is highly recommended that any GA implemented to maximize the fitness score should implement these contracts.

The main GA runner

Preconditions

1. `kwargs` should be supplied
2. `kwargs` is a dict mapping argument names (strings) to argument values
3. The maximum number of generations allowed is greater than 0

Postconditions

1. `kwargs` should not be changed
2. **At least one of the following two conditions must hold**
    (a) the fitness of the fittest individual (being returned) is at least `targetscore`
    (b) the current generation count is equal to the maximum number of generations allowed
3. the maximum number of generations allowed is greater than 0

Individual.py

The following contracts must be followed for any implementation of the `Individual` class

Individual.__hash__(self, other)

Preconditions

None

Postconditions

1. an `int` should be returned
2. `self` should not be changed

In addition to these, the current implementation has the following methods implemented:

Individual.__eq__(self, other)

Preconditions

1. `other` should be an instance of `Individual`

Postconditions

1. `other` should not be changed
2. `self` should not be changed

Individual.__len__(self)

Preconditions

None

Postconditions

1. `self` should not be changed

Individual.__setitem__(self, index, obj)

Preconditions

1. **Exactly one of the following two conditions must be satisfied:**
   (a) 0 <= `index` <= len(self.chromosomes)
   (b) len(self.chromosomes)*-1 >= index >= -1

Postconditions

1. The object at `self.chromosomes[index]` should be `obj`

Individual.__contains__(self, chromosome)

Preconditions

None

Postconditions

1. `self` should not be changed
2. `chromosome` should not be changed

Individual.__repr__(self)

Preconditions

None

Postconditions

1. `self` should not be changed

Individual.append(self, chrom)

Preconditions

None

Postconditions

1. The length of `self.chromosomes` should be increased by exactly 1
2. The last chromosome in `self.chromosomes` should be `chrom`

Individual.count(self, sub, chrom)

Preconditions

None

Postconditions

1. `self` should not be changed

population.py

The following contracts are applied to the functions in population.py

genPop(N, chromGenfuncs, chromGenParams)

Preconditions

1. N >= 0
2. `chromGenfuncs` is a list
3. Every entry in `chromGenfuncs` is a function
4. `chromGenParamss` is a list
5. The lengths of `chromGenfuncs` and `chromGenParams` are equal

Postconditions

1. The inputs are unchanged
2. Function returns a list
3. The length of the returned list is `N`
4. The returned list contains exactly 1 of each item i.e. no two items in the returned list are equal

genCharsChrom(l, chars)

Preconditions

1. `l` is an integer
2. `chars` is an instance of some class that implements `__getitem__`
3. `chars` is an instance of some class that implements `__len__`
4. `len(chars)` is greater than 0

Postconditions

1. The inputs are unchanged
2. Function returns a list
3. The length of the returned list is `l`
4. Every element in the returned list exists in `chars`

genTour(numCities)

Preconditions

1. `numCities` is an integer

Postconditions

1. The inputs are unchanged
2. Function returns a list
3. The length of the returned list is `numCities`
4. Every element in the returned list exists exactly once in the returned list.

score.py

`score(p, scorefuncs, scorefuncparams, SCORES)`

Preconditions

1. `p` is an instance of `Individual`
2. `scorefuncs` is a list of functions
3. `scorefuncparams` is a list of tuples
4. The lengths of `scorefuncs` and `scorefuncparams` are equal
5. `SCORES` is a dictionary

Postconditions

1. The inputs are unchanged
2. `p` is in `SCORES`
3. **Exactly one of the following two conditions are met:**
    (a) `p` was in `SCORES` before this function was called and the number of entries in `SCORES` has not changed
    (b) `p` was not in `SCORES` before this function was called and the number of entries in `SCORES` has increased by exactly 1

`scoreOnes(p)`

Preconditions

1. `p` is list
2. All elements in `p` are strings of length exactly 1
3. All elements in `p` are either '0' or '1'

Postconditions

1. `p` is unchaged
2. An integer is returned
3. The value returned is at least 0

scoreTSP(tour, DIST)

post: isinstance(__return__, float)

post[tour, DIST]: __old__.tour == tour __old__.DIST == DIST

Preconditions

1. `tour` is a list
2. `DIST` is a dictionary
3. All elements in `tour` are integers
4. All keys in `DIST` are integers
5. All values in `DIST` are dictionaries
6. Every key in every value of `DIST` is an integer
7. Every value in every value of `DIST` is a float

Loop Invariant

1. `answer` (the value to be returned is at most 0 and monotonously decreases

Postconditions

1. The inputs are unchanged
2. The function returns a float

getRouletteWheel(pop, SCORES)

Preconditions

1. `pop` is a list of instances of `Individual`
2. `SCORES` is a dictionary
3. Every element in `pop` is a key in `SCORES`

Postconditions

1. The inputs are unchanged
2. A list of 3-tuples of type (Individual, float, float) is returned
3. The length of the returned list is equal to the length of `pop`
4. The first element of every tuple in the returned list exists in `pop`
5. The second float is smaller than the third float in every tuple in the returned list

rouletteWheelSelect(wheel, s=None)

Preconditions

1. **`wheel` is a list of 3-tuples which satisfy all the following conditions**
    (a) The first element is an instance of `Individual`
    (b) The last two elements are floats
    (c) The first float is smaller than the second
2. **Exactly one of the following two conditions are met:**
    (a) `s` is a float
    (b) `s` is None

Postconditions:

1. The inputs are unchanged
2. An instance of `Individual` is returned

tournamentSelect(pop, T, w, n, scorefunc, scoreparams)

Preconditions

1. `pop` is a list of instances of `Individual`
2. `T` is an integer
3. `w` is an integer
4. `n` is an integer
5. `w` is at most `n`
6. `n%w` is exactly 0
7. `n` is at most `T`
8. `scoreparams` is a tuple

Postconditions

1. The inputs are unchanged
2. A list of `n` instances of `Individual` is returned

crossover.py

The following contracts are implemented for the crossover functions.

crossOnes(p1, p2, chrom)

Preconditions

1. `p1` and `p2` are instances of `list`

Postconditions

1. The inputs are unchaged
2. A tuple of two instances of `list` is returned
3. **Each list in the return tuple satisfies the following conditions:**
    (a) each element in the list exists in either `p1` or `p2` or both.

injectionco(p1, p2, chrom)

Preconditions

1. `p1` and `p2` are instances of `list`
2. The length of *p1* is exactly equal to the length of `p2`
3. `p1` is a permutation of [0... `len(p1)-1`]
4. `p2` is a permutation of [0... `len(p2)-1`]

Postconditions

1. The inputs are unchaged
2. A new object is returned of type `list`
3. The length of the returned list is exactly equal to the length of `p1` (and therefore of `p2` as well)
4. The function returns a permutation i.e. all elements in the returned list occur exactly once

twoChildCrossover(p1,p2, crossfuncs, crossparams)

Preconditions

1. `p1` and `p2` are instances of `Individual`
2. `p1` and `p2` are of exactly equal length
3. The number of elements in `crossfuncs` is exactly equal to the length of `p1` (and therefore of `p2`)
4. The number of elements in `crossfuncs` is exactly equal to the number of elements in `crossparams`
5. Every element in `crossparams` is a tuple

Postconditions

1. The inputs are unchanged
2. A tuple of two elements of type `Individual` is returned
3. Each of the returned children has the same number of chromosomes as the parents
4. Each chromosome in each of the children has the same length as the corresponding chromosome of both parents

oneChildCrossover(p1,p2, crossfuncs, crossparams)

Preconditions

1. `p1` and `p2` are instances of `Individual`
2. `p1` and `p2` are of exactly equal length
3. The number of elements in `crossfuncs` is exactly equal to the length of `p1` (and therefore of `p2`)
4. The number of elements in `crossfuncs` is exactly equal to the number of elements in `crossparams`
5. Every element in `crossparams` is a tuple

Postconditions

1. The inputs are unchanged
2. A tuple of one element of type `Individual` is returned
3. The returned child has the same number of chromosomes as the parents
4. Each chromosome in the child has the same length as the corresponding chromosome of both parents

muatation.py

mutateSingleAllele(p, chrom, chars)

Preconditions

1. `p` is an instance of `Individual`
2. `chrom` is an integer
3. The value of each gene in the `chrom` th chromosome of `p` exists in `chars`
4. **Exactly one of the following two conditions must be satisfied:**
   (a) 0 <= `index` <= len(self.chromosomes)
   (b) len(self.chromosomes)*-1 >= index >= -1

Postconditions

1. The inputs are unchanged
2. A new instance of `Individual` is returned
3. The `chrom` th chromosome of the returned individual is not equal to the `chrom` th chromosome of `p`
4. All other chromosomes of the returned individual are exactly the same as the corresponding chromosome of `p`

swapmut(p, chrom)

Preconditions

1. `p` is an instance of `Individual`
2. `chrom` is an integer
3. **Exactly one of the following two conditions are satisfied:**
   (a) 0 <= `chrom` <= `len(p.chromosomes)`
   (b) `len(self.chromosomes)*-1` >= `index` >= -1

Postconditions

1. The inputs are unchaged
2. An instance of `Individual` is returned
3. All values in the `chrom` th chromosome of `p` are present in the `chrom` th chromosome of the output individual
4. The `chrom` th chromosomes of the output individual and `p` are not equal
5. There are exactly two genes in the `chrom` th chromome of `p` and the returned individual, whose values differ

revmut(p, chrom)

Preconditions

1. `p` is an instance of `Individual`
2. `chrom` is an integer
3. **Exactly one of the following two conditions are satisfied:**
   (a) 0 <= `chrom` <= `len(p.chromosomes)`
   (b) `len(self.chromosomes)*-1` >= `index` >= -1

Postconditions

1. The inputs are unchaged
2. An instance of `Individual` is returned
3. All values in the `chrom` th chromosome of `p` are present in the `'chrom` th chromosome of the output individual
4. The `chrom` th chromosomes of the output individual and `p` are not equal

shufflemut(p, chrom)

post[p, chrom]: __old__.p == p __old__.chrom == chrom isinstance(__return__, Individual) __return__.chromosomes[chrom] != p.chromosomes[chrom] forall(p.chromosomes[chrom], lambda e: e in __return__.chromosomes[chrom]) forall(__return__.chromosomes[chrom], lambda e: e in p.chromosomes[chrom])

Preconditions

1. `p` is an instance of `Individual`
2. `chrom` is an integer
3. **Exactly one of the following two conditions are satisfied:**
   (a) 0 <= `chrom` <= `len(p.chromosomes)`
   (b) `len(self.chromosomes)*-1` >= `index` >= -1

Postconditions

1. The inputs are unchaged
2. An instance of `Individual` is returned
3. All values in the `chrom` th chromosome of `p` are present in the `'chrom` th chromosome of the output individual
4. The `chrom` th chromosomes of the output individual and `p` are not equal
5. The length of the `chrom` th chromosome of the returned individual is exactly equal to the length of the `chrom` th chromosome of `p`

CHAPTER 13

Download the PDF

lorem ipsum

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

G

R

S

T