pynetics

Release 0.0.1

Jun 28, 2020

General

Welcome to the official documentation of Pynetics, a free and open source library designed to work and experiment with evolutionary computation algorithms. If you are new to this documentation, we recommend you to check the Getting started section to get an overview of what this library has to offer.

This library is by no means finished (far from it), and I hope it never is. Its goal is to continuously integrate more and more functionality from the field of evolutionary computation, ranging from well-established solutions to state-of-the-art advances.

The main documentation for the site is organized into the following sections:

CHAPTER 1

General

1.1 Introduction

Welcome to the official documentation of Pynetics, a free and open source library designed to work and experiment with Evolutionary Computation (EC) algorithms.

This page gives a broad presentation of the library and how to work with it, so it is a great starting point if you are a beginner.

1.1.1 Before you start

The *Examples* page lists some EC problems solved with the library primitives. We consider it a good approach to learn what can be accomplished with it and what are the main components available.

But if you have any trouble or suggestion, the best way to find help is through one of the various *Community channels*.

1.1.2 About Pynetics

Pynetics is an effort to write a free and open source community-driven CE library. This means that its development is open to everyone who wants to collaborate, as long as its purposes and its community are respected.

1.1.3 About the documentation

This documentation is being continuously written, corrected and edited. The main language is reStructuredText compiled into a static website/offline document using Sphinx and ReadTheDocs tools.

1.1.4 Organization of the documentation

TODO

1.2 Frequently asked questions

1.2.1 What can I do with Pynetics?

Pynetics is Free and Open-Source Software available under the OSI-approved MIT license. This means it is free as in "free beer".

In essence, you can do whatever you want as long as you include the original copyright and license notice in any copy of the software/source.

In addition, all the documentation is licensed under the permisive Creative Commons Attribution 4.0 (CC-BY 4.0) license, with attribution to "Alberto Díaz-Álvarez and the Pynetics community."

For full details, take a look at the LICENSE.md file in the Pynetics repository.

1.2.2 I would like to contribute! How can I get started?

Awesome! In fact, it is one of the main reasons why this library is free, to continuously improve with your ideas and innovations and give back to the community.

There are many different ways to collaborate: developing new features, improving documentation, adding examples, identifying bugs, ... A very interesting starting point are the issues; you can create one related to the missing or failed functionalities or examples, or you can also choose the one that most appeals to you to fork the repo, modify it and submit a Merge Request with your changes (TODO CREATE A CONTRIBUTING AND A CODE OF CONDUCT FILES AND LINK TO THEM).

You also have the option to enter our Discord Server, to comment on your concerns or on what you could contribute. You can even help us to identify how to better organize the project. Any help is welcome.

CHAPTER 2

Getting started

2.1 Examples

Each of the following examples are located in the *examples* directory under the root directory of the library.

2.1.1 Ones counting

This is an example to show how to create a basic Genetic Algorithm. the idea is to achieve an all-1s sequence from a population of random binary sequences.

With this example it is possible to explore the basic elements of a genetic algorithm (e.g. genotypes, operators, etc.) and the existing components in the Pynetics library to create and use them.

In the example, we've established the genotype size via the next sentence:

```
TARGET_LEN = 50
```

We don't really need to code almost anything else. Pynetics has a set of components for list-based genetic algorithms that can be used for this kind of problems. One of them is the ListGenotype, an implementation of a Genotype that behaves as a list of genes. For now, that's all we need to know.

Fitness

The dependent aspect of our problem is the answer to the question: how can we tell our algorithm how good an individual is? This is where the fitness comes into play.

The fitness can be any function that receives an object (representing an individual) and returns a float value where, the higher the value, the fitter the individual. We have implemented ours as follows.

```

```

def fitness(phenotype): return sum(phenotype) / len(phenotype)

This function returns a value that goes from 0 to 1 being 0 the worst that evolution has been able to create and 1 totally fit. In fact, any function that admits an object of the Genotype class as an argument and returns a float whose value is larger the fitter the individual is, would be sufficient.

Algorithm configuration

This is actually the most we're going to program specifically for our problem. The rest will be to configure the genetic algorithm to perform the iterative process that it consists of and provide us with a solution.

We're going to write down all the settings and comment on them in parts:

```
def fitness(genotype):
    return sum(genotype.genes) / len(genotype)

ga = GeneticAlgorithm(
    population_size=10,
    initializer=AlphabetInitializer(size=TARGET_LEN, alphabet=BINARY),
    stop_condition=FitnessBound(1),
    fitness=fitness,
    selection=Tournament(3),
    recombination=(random_mask, 1.0),
    mutation=(RandomGene(BINARY), 1 / TARGET_LEN),
    replacement=(high_elitism, 1.0),
)
```

`GeneticAlgorithm` is the main class we're going to use. It represents a genetic algorithm configured with a set of settings to drive its behavior. Those settings are mandatory and are specified here:

- `population_size`: How many gentoypes will exist in our population at a time.
- `initializer': The object in charge of creating individuals when needed (almost always in the initialization step of the genetic algorithm), where as many as specified in the population size will be created. In our case we will use an :code:`AlphabetInitializer`, which will create individuals using a `BINARY` alphabet of the specified length.
- `stop_condition`: When the algorithm should stop. In our case, we're using an stopper that deals with the genotype fitness.
- `fitness`: Which function we'll use to evaluate the individuals. This is the one we defined before.
- `selection`: Which operator the algorithm will use in order to select individuals from the population. We use a *tournament selection* schema with 3 random genotypes.
- `recombination` and `recombination_probability`: A recombination algorithm and the actual probability for the individuals to actually match. In this case, we're using a *random mask* recombination schema with a probability of 1 (i.e. there'll always be a match).
- `mutation` and `mutation_probability`: Like the previous two, but this time with the mutation. In our case we'll' work with a *random_gene* schema with a low probability.
- `replacement`: Which replacement schema to use and the replacement ratio (percentage of how many individuals are needed in the offspring to actually replace populations. We will use a *high elitism* schema with a 100% replacement rate (the offspring will have the same length as the original population.

Running the algorithm

Once the algorithm has been configured, we just need to run it:

```
history = ga.run()
```

It may take a while (I hope not, at least this example), but one it finishes it will return a history object with information about the execution. In our case, we recover the best genotype of the last generation and print it, along with the generation it appeared and its fitness.

```
best = history.data['Best genotype'][-1]
print(history.generation, best, best.fitness())
```

And that's all! I hope you were under the impression of how to work with the library.

2.1.2 Ones counting with callbacks

In this example, we introduce the Callback instances. When creating a new instance of a GeneticAlgorithm, the user has the option of providing a list of callbacks, objects that are called whenever an event is triggered. The previous *Ones counting* examples will be used as base for this one.

There is a class in the library called Callback, which defines the possible events, but actually any object with the required methods is valid as a callback. The methods, corresponding to the events, are self-explanatory, and are the following:

- `on_algorithm_begins(self, ga: api.EvolutiveAlgorithm)`
- `on_algorithm_ends(self, ga: api.EvolutiveAlgorithm)`
- `on_step_begins(self, ga: api.EvolutiveAlgorithm)`
- `on_step_ends(self, ga: api.EvolutiveAlgorithm)`

Defining a callback

Our example defines a callback called `MyCallback` which inherits from the `Callback` helper class, redefining the four methods:

```
class MyCallback(Callback):
    def on_algorithm_begins(self, g):
        print('Start algorithm')

    def on_step_begins(self, g):
        print('Starting step ... ', end='')

    def on_step_ends(self, g):
        print('generation: {}\tfitness: {:.2f}\tIndividual: {}'.format(
            g.generation,
            g.best().fitness(),
            g.best(),
        ))

    def on_algorithm_ends(self, g):
        print('End algorithm')
```

Algorithm configuration

The configuration is the same as the previous example with the difference that we've added a new argument, `callback` with a list of objects to be used as callbacks; in our case, only one, an instance of `MyCallback`:

```
ga = GeneticAlgorithm(
    # ...
    # Stuff
    # ...
    callbacks=[MyCallback()],
)
```

Then, during the execution of the algorithm, each time an event is triggered (e.g. a new algorithm iteration or step begins), the corresponding methods from each callback in the list (e.g. `on_step_begins`) are called.

2.1.3 Ones counting finishing abruptly

The callbacks in a genetic algorithm allows us to alter its behaviour while it is running. This means we can change any operator depending on other factors (e.g. altering mutation probability as a function of diversity).

In this example, we're gonna replicate the previous *Ones counting with callbacks*, but it'll be stopped once a series of `m` consecutive ones are generated.

The new callback

Lacking a better name, we decided to call the new callback `StopBecauseIAmWorthIt`. It'll be implemented as follows:

```
class StopBecauseIAmWorthIt(Callback):
    """Stops if there are n consecutive 1's in the genotype"""

    def __init__(self, m):
        self.m = m
        self.ones = '1' * self.m

    def on_step_ends(self, g):
        for genotype in g.population:
            if self.ones in ''.join(map(str, genotype)):
                print(f"Stopping because we found {self.m} consecutive 1's")
                g.stop()
```

In short, it will stop as soon as at least one genotype in the population has `m` consecutive ones. This'll be done by calling the genetic algorithm `stop` method.

Algorithm configuration

The configuration is the same as the previous example, but using an instance of our new callback:

```
ga = GeneticAlgorithm(
    # ...
    # Stuff
    # ...
    callbacks=[StopBecauseIAmWorthIt(m=42)]
)
```

The callback is configured to alter the genetic algorithm behavior once 42 consecutive ones are found in any genotype.

2.1.4 Hello world

In the previous *ones counting examples* we made use of a binary alphabet to encode our genotypes.

Although a binary alphabet can encode virtually any problem with finite alphabets, sometimes more complex alphabets are more useful. In this example we will try to achieve a predefined phrase with an alphabet tailored to our problem.

The problem is as follows: Let's assume we have an unknown sentence consisting of letters, numbers, punctuation marks and spaces, and we want to build an algorithm that finds that sentence.

We just need to program a new fitness function and provide a genotype initializer with our alphabet. But the

Fitness

Our target sentence will be in the `TARGET` variable, and will be of length `TARGET_LEN`. For the fitness we will compare the phenotype (i.e. the resulting sentence from the genotype) with the target sentence using the Hamming distance. The smaller the distance, the closer the sentence is to the best solution, thus the higher the fitness.

An implementation for this fitness could be as follows:

```
def fitness(phenotype):
    """The fitness will be based on the hamming distance error."""
    # Derive the phenotype from the genotype
    sentence = ''.join(str(x) for x in phenotype)
    # Compute the error of this solution
    error = len([i for i in filter(
    lambda x: x[0] != x[1], zip(sentence, TARGET)
    )])
    # Return the fitness according to that error
    return 1 / (1 + error)
```

This is not the only implementation, but it is simple enough to see what's happening. We are assuming that our genotypes are composed of genes belonging to an specific alphabet. Let's create the initializer that will generate those genotypes.

Alphabet and the genotype initializer

In the previous examples, we used the class `AlphabetInitializer` with a binary alphabet. Actually, this class works with any alphabet, provided that it consists of a finite number of elements.

We are going to use exactly the same object, but with a different alphabet, the one that encodes our solution:

```
alphabet = Alphabet(
    genes=string.ascii_letters + string.punctuation + ' '
)
```

Now we have an `Alphabet` that codifies our solutions. We can use it to create our initializer:

```
initializer = AlphabetInitializer(size=TARGET_LEN, alphabet=alphabet)
```

Being `TARGET_LEN` the length of the sentence we'll try to discover and `alphabet` the alphabet that codifies our solutions.

Algorithm configuration

Let's configure out algorithm. As we did before, it is enough to use the default operators.

```
ga = GeneticAlgorithm(
    population_size=10,
    initializer=AlphabetInitializer(size=TARGET_LEN, alphabet=alphabet),
    stop_condition=FitnessBound(1),
    fitness=fitness,
    selection=Tournament(4),
    recombination=(one_point_crossover, 1.0),
    mutation=(RandomGene(alphabet), 1 / TARGET_LEN),
    replacement=(high_elitism, 1.0),
    callbacks=[MyCallback()]
)

ga = GeneticAlgorithm(
    # ... Stuff ...
    initializer=AlphabetInitializer(size=TARGET_LEN, alphabet=alphabet),
    # ... More stuff ...
    mutation=(RandomGene(alphabet), 1 / TARGET_LEN),
    # ... Even more stuff ...
)
```

2.1.5 Two decks problem

In the two decks problem there's a deck of N cards, numbered from 1 to N, and they have to be divided in two decks in such a way that the cards in one pile sum as close as possible to `I` and the cards in the other multiply as close as possible to `J`.

We can use a ListGenotype with a binary alphabet, so a length of N represents N cards ranging from 1 to N, and each gene value (i.e. 0 or 1) maps that card to one deck (i.e. 1 or 2 respectively).

Customizing the phenotype

We're gonna introduce a new concept. A `ListGenotype` is enough, yes, but it will be nice to vary it's genotype. We dont care about the internal representation; we only want the actual individual, that is, the two decks.

It requires two things; first, overriding the `phenotype` method (obviously), and second, telling to the initializer that the class of the `ListGenotype` instances is not the standard, but is the new one created by us. So let's get to it.

First, our `ListGenotype` subclass:

```
class CardsGenotype(ListGenotype):
    def phenotype(self):
        """The phenotype will be a tuple of two decks containing a list
        of the cards included on that decks."""
        # Those positions marked as 0 will belong to the deck 1 (sum)
        deck1 = [i for (i, g) in enumerate(self, start=1) if g == 0]
        # Those positions marked as 1 will belong to the deck 2 (productS)
        deck2 = [i for (i, g) in enumerate(self, start=1) if g == 1]

        return deck1, deck2
```

Once we have our class (and it's a `ListGenotype` subclass), we just need to create the initializer advising it to use our custom class instead of the default `ListGenotype`.

```
initializer = AlphabetInitializer(
        size=N, alphabet=alphabet.BINARY, cls=CardsGenotype
    )
```

We can now continue with the fitness implementation.

Fitness

The idea is to reach two targets. In the example are specified as two variables, `TARGET_I` and `TARGET_J`:

```
TARGET_I = 36   # Sum target
TARGET_J = 360  # Product target
```

In our fitness, we will compute the error to each target (using the proportional difference) and then add it. Maybe there is a better solution, but this one does the trick.

```
def fitness(phenotype):
    deck1, deck2 = phenotype

    error1 = abs((sum(deck1) - TARGET_I) / TARGET_I)
    error2 = abs((reduce(operator.mul, deck2, 1) - TARGET_J) / TARGET_J)

    error = error1 + error2

    # Return the fitness according to that error
    return 1 / (1 + error)
```

Check the first line of the function. Do you see how the custom phenotype is being used? Nice, now we're prepared to configure and run the algorithm as usual.

Algorithm configuration

Let's configure out algorithm. As we did before, it is enough to use the default operators, but this time telling to the `AlphabetInitializer` that the `ListGenotype` class to use is out implementation.

```
ga = GeneticAlgorithm(
    # ... Stuff ...
    initializer=AlphabetInitializer(
        size=N, alphabet=alphabet.BINARY, cls=CardsGenotype
    ),
    # ... More stuff ...
)
```

CHAPTER 3

Development

TBD

CHAPTER 4

Community

4.1 Community channels

So, where can you ask for help in case something doesn't work?

For now, there's only two channels of communication, the Discord Server and via email (yes, direct contact to me). I hope that in the future there will be a cohesive community around the library. I look forward to your comments and suggestions!

CHAPTER 5

Library reference

5.1 pynetics

5.1.1 pynetics package

Subpackages

pynetics.list package

Submodules

pynetics.list.alphabet module

TODO TBD…

class pynetics.list.alphabet.**Alphabet**(**, genes: Iterable[Any]*)
Bases: object

Alleles based on a list of equiprobable elements.

Although it can be initialized by repeated genes, only one of each will be stored.

genes
The genes that conform this alphabet.

get(*n: int = 1*, *rep: bool = True*) → Union[Any, Sequence[Any]]
Returns 1-to-n random values among all the possible values.

The method will return either a random allele from the whole genetic pool or a sequence of alleles, with or without repetition.

Parameters

- **n** – The number of genes to return. Defaults to 1.

- **rep** – If n > 1, True means repeated alleles may be returned, whereas False means no repeated alleles will be returned. Defaults to True.

Returns An allele in case n = 1 or a sequence of alleles in case n > 1.

Raises ***MoreGenesRequiredThanExisting*** – If n > len(genetic pool) and rep is False (if we don't allow repetition, it is impossible to return more values than there are.

pynetics.list.bin module

Specific implementations for binary alphabet based algorithms.

pynetics.list.bin.**generalised_crossover**(*parent1: pynetics.list.genotype.ListGenotype*, *parent2: pynetics.list.genotype.ListGenotype*) → Tuple[pynetics.list.genotype.ListGenotype, pynetics.list.genotype.ListGenotype]

Progeny obtained by crossing genotypes as if they where integers.

This recombination algorithm is expected to be used with binary genotypes of the same length. It may work with other kind of individuals, but the outcome is unpredictable.

Parameters

- **parent1** – One of the genotypes.
- **parent2** – The other.

Returns A tuple with the progeny.

pynetics.list.diversity module

Implementations of different diversity algorithms for genotypes represented by lists.

class pynetics.list.diversity.**DifferentGenesInPopulation**(*alphabet: pynetics.list.alphabet.Alphabet*)

Bases: object

Diversity based on the appearance of different genes regardless their position.

The value is computed as follows. For each gene position, a value of M (the different appearing genes) is computed. Then, all the values are sum added and the divided by Y = N * L where L is the length of the individual and N the number of possible alleles. The value then is expected to belong to the interval [0, 1], where 0 is no diversity at all and 1 a completely diverse population.

It is expected for the genotypes to have the same length.

pynetics.list.diversity.**average_hamming**(*genotypes: Sequence[pynetics.list.genotype.ListGenotype]*) → float

Average of each hamming loci distances.

The diversity will be computed as follows:

1. **Calculate all the hamming distances between each pair of** genotypes.
2. Divide between the length of the genotypes.

It is assumed that all the genotypes have the same length; if not, the diversity will be computed as if all the genotypes had the same length (the minimum among them).

Parameters **genotypes** – A sequence of individuals from which obtain the diversity.

Returns A float value representing the diversity.

pynetics.list.exception module

Definition specific errors for list based genetic algorithms.

exception pynetics.list.exception.**AlphabetError**
Bases: *pynetics.exception.PyneticsError*

Errors related to the Alphabet classes.

exception pynetics.list.exception.**EmptyListGenotype**
Bases: *pynetics.exception.RecombinationError*

A genotype with at least one gene was needed, but it was zero.

exception pynetics.list.exception.**NotEnoughSymbolsInAlphabet**(*, *expected: int*, *real: int*)
Bases: *pynetics.list.exception.AlphabetError*

The alphabet hasn't as many symbols as needed.

exception pynetics.list.exception.**TooFewGenes**(*, *genes: Iterable[Any]*)
Bases: *pynetics.list.exception.AlphabetError*

Very few genes to fill up the alphabet.

pynetics.list.genotype module

Implementations for list based genotypes.

class pynetics.list.genotype.**ListGenotype**(*, *genes: Optional[Iterable[Any]] = None*)
Bases: *pynetics.api.Genotype*, collections.abc.MutableSequence

A list based genotype, used for various implementations.

It also behaves as a MutableSequence (provides __init__, __len__, __getitem__, __setitem__, __delitem__, and insert), so be used too almost like a drop-in replacement for a list.

insert(*index: int*, *value: Any*)
Inserts an element in the given index.

If the given index is greater or equal to the length of the genotype, the value is inserted at the end of it, regardless the length of it.

The elements to the right are shifted one position.

Parameters

- **index** – The position to insert the value.
- **value** – Which value insert.

phenotype() → List[Any]
Returns the list of genes conforming this genotype.

Returns An ordered list with the elements conforming this genotype.

pynetics.list.initializer module

API implementations specific of list based genotypes problems.

class pynetics.list.initializer.**AlphabetInitializer**(*, *size: int*, *alphabet: pynetics.list.alphabet.Alphabet*, *cls: Optional[Type[pynetics.list.genotype.ListGenotype]] = None*)

Bases: *pynetics.list.initializer.ListGenotypeInitializer*

Initializer for ListGenotype based on an arbitrary alphabet.

create() → pynetics.list.genotype.ListGenotype
Generates a new random genotype.

Returns A new ListGenotype instance.

class pynetics.list.initializer.**IntervalInitializer**(*, *size: int*, *lower: Union[int, float]*, *upper: Union[int, float]*, *cls: Optional[Type[pynetics.list.genotype.ListGenotype]] = None*)

Bases: *pynetics.list.initializer.ListGenotypeInitializer*

Initializer for genotypes whose genes belong to an interval.

This is an abstract class that comprises the common behavior for genotypes of either integer or real genes.

create() → pynetics.list.genotype.ListGenotype
Generates a new random genotype.

Returns A new genotype instance.

get_value_from_interval() → Union[int, float]
Generates a new value that belongs to the interval.

Returns A value for the specified interval at initialization.

class pynetics.list.initializer.**ListGenotypeInitializer**(*, *size: int*, *cls: abc.ABCMeta*)

Bases: *pynetics.api.Initializer*

Common behaviour between initializers.

class pynetics.list.initializer.**PermutationInitializer**(*, *size: int*, *alphabet: pynetics.list.alphabet.Alphabet*, *cls: Optional[Type[pynetics.list.genotype.ListGenotype]] = None*)

Bases: *pynetics.list.initializer.ListGenotypeInitializer*

TODO TBD...

create() → pynetics.list.genotype.ListGenotype
Generates a new random genotype.

Returns A new ListGenotype instance.

pynetics.list.int module

TODO TBD...

class pynetics.list.int.**IntegerIntervalInitializer**(*, *size: int*, *lower: Union[int, float]*, *upper: Union[int, float]*, *cls: Optional[Type[pynetics.list.genotype.ListGenotype]] = None*)

Bases: *pynetics.list.initializer.IntervalInitializer*

Initializer for int based ListGenotype instances.

get_value_from_interval() → int

Generates a new value that belongs to the interval.

This value will be an integer value.

Returns A value for the specified interval at initialization.

class pynetics.list.int.**RangeCrossover**(*, *lower: int*, *upper: int*)

Bases: object

Offspring is obtained by crossing individuals gene by gene.

For each gene, the interval of their values is calculated. Then, the difference of the interval is used for calculating the new interval from where to pick the values of the new genes. First, a value is taken from the new interval. Second, the other value is calculated by taking the symmetrical by the center of the range.

It is expected for the genotypes to have the same length. If not, the operation works over the first common genes, leaving the rest untouched.

pynetics.list.mutation module

TODO TBD...

class pynetics.list.mutation.**RandomGene**(*alphabet: pynetics.list.alphabet.Alphabet*, *same: bool = False*)

Bases: object

Mutates the genotype by changing some genes values.

For each gene the mutation probability will be check and, if a mutation is requested, that gene will be replaced by a random one extracted from the alphabet.

genotype : aacgaata alphabet : (a, c, t, g) mutate in : 2, 7 ———— mutated : abcdaaba

pynetics.list.mutation.**swap_genes**(*p_mutation: float*, *genotype: pynetics.list.genotype.ListGenotype*) → pynetics.list.genotype.ListGenotype

Mutates the genotype by swapping two random genes.

For each gene, a random value will be compared against the mutation probability and, if the value is lower, the mutation will take place and its value will be swapped with the value of another random gene in the genotype.

For example, if the mutation occurs in the gene located in position 3 and it has to be swapped with the one in position 5, then:

genotype : 12345678 positions : 3, 5 ———— mutated : 12365478

Parameters

- **p_mutation** – The probability of mutation.
- **genotype** – The genotype to be mutated.

Returns A new mutated genotype. If no mutation was performed, the same genotype instance is returned.

pynetics.list.real module

TODO TBD...

class pynetics.list.real.**FlexibleRecombination**(*, *lower: float*, *upper: float*, *alpha: float*)

Bases: object

TODO TBD...

class pynetics.list.real.**RealIntervalInitializer**(*, *size: int*, *lower: Union[int, float]*, *upper: Union[int, float]*, *cls: Optional[Type[pynetics.list.genotype.ListGenotype]] = None*)

Bases: *pynetics.list.initializer.IntervalInitializer*

Initializer for int based ListGenotype instances.

get_value_from_interval() → float

Generates a new value that belongs to the interval.

This value will be an integer value.

Returns A value for the specified interval at initialization.

pynetics.list.real.**plain_recombination**(*parent1: pynetics.list.genotype.ListGenotype*, *parent2: pynetics.list.genotype.ListGenotype*) → Tuple[pynetics.list.genotype.ListGenotype, pynetics.list.genotype.ListGenotype]

TODO TBD...

Parameters

- **parent1** – One of the genotypes.
- **parent2** – The other.

Returns A tuple with the progeny.

pynetics.list.recombination module

Recombination algorithms for list-based genotypes.

class pynetics.list.recombination.**NPivot**(*, *num_pivots: int*)

Bases: object

Progeny obtained by mixing parents with N random pivot points.

The process is as follows. N random points belonging to the [1, L-1] interval (being L the size of the genotypes) is selected. Then, one by one, the contents of both individuals are interchanged each time a new pivot point is reached. For example:

pivots : 3, 5 parents : XXXXXXXX, OOOOOOOO ———— children : XXXOOXXX, OOOXXOOO

The method implementation has been slightly modified to allow the recombination of different length genotypes. In this case, L will be the length of the shortest genotype.

Also, if the specified pivots number is greater than L-1, then L-1 will be used as the new N.

pynetics.list.recombination.**pmx**(*parent1: pynetics.list.genotype.ListGenotype*, *parent2: pynetics.list.genotype.ListGenotype*) → Tuple[pynetics.list.genotype.ListGenotype, pynetics.list.genotype.ListGenotype]

TODO TBD...

Parameters

- **parent1** – One of the genotypes.
- **parent2** – The other.

Returns A tuple with the progeny.

pynetics.list.recombination.**random_mask**(*parent1: pynetics.list.genotype.ListGenotype*, *parent2: pynetics.list.genotype.ListGenotype*) → Tuple[pynetics.list.genotype.ListGenotype, pynetics.list.genotype.ListGenotype]

Progeny is created by using a random mask.

The random mask is generated to determine which genes are switched between genotypes. For example:

random mask : 00100110 parents : XXXXXXXX, OOOOOOOO ——— children : XXOXXOOX, OOXOOXXO

The method implementation has been slightly modified to allow the recombination of different length genotypes. In this case, the mask will only affect to the minimum length, leaving the rest of the genes untouched.

Parameters

- **parent1** – One of the genotypes.
- **parent2** – The other.

Returns A tuple with the progeny.

Module contents

Submodules

pynetics.algorithm module

Implementation of different Evolutionary Computation algorithms.

Currently only one implementation (Genetic Algorithm) is provided.

class pynetics.algorithm.**GeneticAlgorithm**(**, population_size: int, initializer: api.Initializer, stop_condition: api.StopCondition, fitness: api.Fitness, selection: api.Selection, replacement: Tuple[api.Replacement, float], recombination: api.Recombination = None, recombination_probability: float = None, mutation: Optional[api.Mutation] = None, mutation_probability: Optional[float] = None, callbacks: Sequence[callback.Callback] = None*)

Bases: *pynetics.api.EvolutiveAlgorithm*

Modular implementation of a canonical genetic algorithm.

Each step the algorithm will generate a new offspring population from the previous one, and then invoke the replacement schema to combine the old and new genotypes in a new population ready to use in a new algorithm step.

The step consists in a loop where the selection, replacement and mutation operators take part. This loop will end when the offspring population is filled.

best() → pynetics.api.Genotype
The best genotype obtained so far.

Returns The best genotype obtained at the moment of calling.

Raise NotInitialized if the population hasn't be created yet.

mutation
TODO TBD

mutation_probability
TODO TBD

on_finalize()
Subclass specific finalization method

on_initialize()
Subclass specific initialization method

recombination
TODO TBD

recombination_probability
TODO TBD

step()
The particular implementation of this genetic algorithm.

The inner working is as follows:

pynetics.api module

Definition of the generic high level API for the library.

The rest of modules and packages will either inherit from these or use them ignoring their implementations.

class pynetics.api.**EvolutiveAlgorithm**(*stop_condition: Callable[[pynetics.api.EvolutiveAlgorithm], bool], callbacks: Optional[Sequence[pynetics.callback.Callback]] = None*)
Bases: object

Base class which defines how a genetic algorithm works.

More than one algorithm may exist so a base class is created to specify the required contract to be implemented by the other classes to work properly.

class EventType
Bases: enum.Enum

The event type fired.

It is used to differentiate which event should be called in the algorithm listeners subscribed to the class.

ALGORITHM_BEGIN = 'on_algorithm_begins'

ALGORITHM_END = 'on_algorithm_ends'

STEP_BEGIN = 'on_step_begins'

STEP_END = 'on_step_ends'

best() → pynetics.api.Genotype
Returns the best genotype in this current generation.

Returns The best genotype.

callbacks

finalize()
Performs the finalization operations.

generation
The current generation in the algorithm.

initialize()
TODO: TBD...

on_finalize()
Subclass specific finalization method

on_initialize()
Subclass specific initialization method

run()
Runs the algorithm.

running()
TODO TBD...

step()
Called on every iteration of the algorithm.

stop()
TODO TBD...

class pynetics.api.**Genotype**
Bases: object

A possible codification of a solution to a problem.

fitness() → float
Returns the fitness of this particular genotype.

It relies on the parameter fitness_function, which is filled by the Population object that manages each population inside a genetic algorithm.

Returns The fitness of this particular genotype

phenotype() → Any
The phenotype resulting from this genotype.

Returns An object representing the phenotype of this genotype.

class pynetics.api.**Initializer**
Bases: object

Responsible for new genotypes generation.

This class is used for both initialization phase (just before the first genetic algorithm step) and genotype creation when needed.

create() → pynetics.api.Genotype
Creates a new genotype.

This method should be overwritten, as it depends completely on the problem codification.

Returns A newly created genotype.

fill(*population: pynetics.api.Population*) → pynetics.api.Population
Fills the population with newly created genotypes.

The creation of those genotypes will be delegated to the create method.

The population passed as argument will be modified. The returned population is nothing but a reference to the same population for fluent API purposes.

Parameters **population** – The population to be filled. Cannot be None.

Returns The same population, but with the new genotypes in it.

class pynetics.api.**Population**(*size: int, fitness: Callable[[Any], float]*)
Bases: collections.abc.MutableSequence, typing.Generic

Manages the pool of genotypes that are solutions of a G.A.

It's a subclass of list, but with some additions.

append(*genotype: pynetics.api.Genotype*) → None
Add a new genotype to this population.

The implementation is delegated to the super class (list) but the population is marked as not sorted and it raises an error in case the maximum size (specified at initialization time) is reached.

Notice that, after calling this method, the genotype object attribute fitness_function will be overriden by the one set in this population object.

Parameters **genotype** – The genotype to add into this population.

Raises **FullError** – in case the population has already reached its full size.

clear()
Removes all the population genotypes.

empty() → bool
Points out if the population is empty or not.

Returns True if the population is empty or false otherwise.

extend(*genotypes: Iterable[pynetics.api.Genotype]*) → None
Add a new bunch of genotypes to this population.

The implementation is delegated to the super class (list) but the population is marked as not sorted and it raises an error in case the maximum size (specified at initialization time) is reached.

Notice that, after calling this method, the genotype object attribute fitness_function will be overriden by the one set in this population object.

Parameters **genotypes** – The genotypes to add into this population.

Raises **FullError** – in case the population has already reached its full size.

full() → bool
Points out if the population is full or not.

Returns True if the population is complete or false otherwise.

insert (*index: int*, *genotype: pynetics.api.Genotype*) → None
Add a new genotype to this population.

The implementation is delegated to the super class (list) but the population is marked as not sorted and it raises an error in case the maximum size (specified at initialization time) is reached.

Notice that, after calling this method, the genotype object attribute fitness_function will be overriden by the one set in this population object.

Parameters

- **index** – The genotype to add into this population.
- **genotype** – The genotype to add into this population.

Raise FullError in case the population has already reached its full size.

pop (*i: int = 0*) → pynetics.api.Genotype
Extracts the object in the specified position.

Parameters **i** – The position. If not specified, it will be position 0.

reverse () → None
Reverse the elements of the list in place.

The implementation is delegated to the super class (list) but the population is marked as not sorted.

sort (***kwargs*) → None
Sorts the genotypes of this population by a function.

After calling this method, the order of the population will be in ascending order, i.e. from lower to higher fitness.

pynetics.callback module

General behaviour of the callbacks called while training.

class pynetics.callback.**Callback**
Bases: object

Base class to listen Genetic Algorithm events during training.

It is useful not only to monitor how the whole genetic algorithm is evolving during training, but also to modify it in real time (i.e. the problem constraints change in real time).

on_algorithm_begins (*ga: pynetics.api.EvolutiveAlgorithm*)
The method to be called when the genetic algorithm starts.

It will be called AFTER initialization but BEFORE the first iteration, including the check against the stop condition.

Parameters **ga** – The genetic algorithm that caused the event.

on_algorithm_ends (*ga: pynetics.api.EvolutiveAlgorithm*)
The method to be called when the genetic algorithm ends.

It will be called AFTER the stop condition has been met.

Parameters **ga** – The genetic algorithm that caused the event.

on_step_begins (*ga: pynetics.api.EvolutiveAlgorithm*)
The method to be called when an iteration step starts.

It will be called AFTER the stop condition has been checked and proved to be false) and BEFORE the new step is computed.

Parameters **ga** – The genetic algorithm that caused the event.

on_step_ends(*ga: pynetics.api.EvolutiveAlgorithm*)
The method to be called when an iteration step ends.

It will be called AFTER an step of the algorithm has been computed and BEFORE a new check against the stop condition is going to be made.

Parameters **ga** – The genetic algorithm that caused the event.

class pynetics.callback.**History**
Bases: *pynetics.callback.Callback*

Keep track of some of the indicators of the algorithm.

This callback is automatically added by the GeneticAlgorithm base class and is returned by its *run* method.

on_algorithm_begins(*ga: pynetics.api.EvolutiveAlgorithm*)
When the algorithm starts, the parameters are reset.

Parameters **ga** – The genetic algorithm that caused the event.

on_step_ends(*ga: pynetics.api.EvolutiveAlgorithm*)
Once a step is finished, some indicators are recorded.

Those indicators include the best genotype instance and the best fitness achieved.

Parameters **ga** – The genetic algorithm that caused the event.

pynetics.exception module

Definition of library specific errors.

exception pynetics.exception.**AllelesError**
Bases: *pynetics.exception.PyneticsError*

exception pynetics.exception.**BoundsCannotBeTheSame**(*value*)
Bases: *pynetics.exception.PyneticsError*

The lower and upper bounds are equal and cannot be.

exception pynetics.exception.**CannotSelectThatMany**(*size: int*, *n: int*)
Bases: *pynetics.exception.SelectionError*

The population has fewer chromosomes than the required.

exception pynetics.exception.**ElementNotFound**
Bases: *pynetics.exception.PyneticsError*

The element was not found where it was supposed to be.

exception pynetics.exception.**EmptyPopulation**(*population_name: str = None*)
Bases: *pynetics.exception.PopulationError*

The population has no individuals.

exception pynetics.exception.**FullPopulationError**(*size: int*)
Bases: *pynetics.exception.PopulationError*

We are trying to add more genotypes than the allowed.

exception pynetics.exception.**GeneticAlgorithmError**
Bases: *pynetics.exception.PyneticsError*

exception pynetics.exception.**MoreGenesRequiredThanExisting**(*req_size: int*, *pool_size: int*)

Bases: *pynetics.exception.AllelesError*

exception pynetics.exception.**NotInitialized**

Bases: *pynetics.exception.GeneticAlgorithmError*

The algorithm hasn't been initialized before start.

exception pynetics.exception.**OffspringSizeBiggerThanPopulationSize**(**, population_size, offspring_size*)

Bases: *pynetics.exception.ReplacementError*

The offspring size cannot be bigger than the population size.

exception pynetics.exception.**PopulationError**

Bases: *pynetics.exception.PyneticsError*

exception pynetics.exception.**PopulationSizesDoNotMatchAfterReplacement**(**, old_size, new_size*)

Bases: *pynetics.exception.ReplacementError*

Population sizes dont match after the replacement operation.

exception pynetics.exception.**PyneticsError**

Bases: `Exception`

Generic class for the errors raised from inside the library.

exception pynetics.exception.**RecombinationError**

Bases: *pynetics.exception.PyneticsError*

Errors related to mating issues.

exception pynetics.exception.**ReplacementError**

Bases: *pynetics.exception.PyneticsError*

Errors related to replacement issues.

exception pynetics.exception.**SelectionError**

Bases: *pynetics.exception.PyneticsError*

Errors related with selection algorithms

exception pynetics.exception.**StopConditionError**

Bases: *pynetics.exception.PyneticsError*

Any error related to stop conditions

exception pynetics.exception.**WrongSelectionSize**(*n: int*)

Bases: *pynetics.exception.SelectionError*

Tried to extract zero or negative genotypes.

pynetics.replacement module

Replacement algorithms.

class pynetics.replacement.**HighElitism**(**, maintain: bool = True*)

Bases: *pynetics.replacement.ReplacementSchema*

Replacement with the fittest among both population and offspring.

Only those best genotypes among both populations will be selected, thus discarding those less fit. This makes this operator extremely elitist.

do(*, *population: pynetics.api.Population*, *offspring: pynetics.api.Population*) → pynetics.api.Population
Executes this replacement.

Parameters

- **population** – The original population.
- **offspring** – The population to replace the original one.

Returns A new Population instance.

class pynetics.replacement.**LowElitism**(*, *maintain: bool = True*)
Bases: *pynetics.replacement.ReplacementSchema*

Replaces the less fit of the population with the fittest of the offspring.

The method will replace the less fit genotypes by the best ones of the offspring. This makes this operator elitist, but at least not much. Moreover, if the offspring size equals to the population size then it's a full replacement (i.e. a generational scheme).

do(*, *population: pynetics.api.Population*, *offspring: pynetics.api.Population*) → pynetics.api.Population
Executes this replacement.

Parameters

- **population** – The original population.
- **offspring** – The population to replace the original one.

Returns A new Population instance.

class pynetics.replacement.**ReplacementSchema**(*, *maintain: bool = True*)
Bases: object

Groups common behavior across all the replacement schemas.

The replacement schema is defined as a class. However, it is enough to implement it a replacement method, i.e. a function that receives two populations (original and offspring) and returns a population resulting from the combination of the previous two.

do(*, *population: pynetics.api.Population*, *offspring: pynetics.api.Population*) → pynetics.api.Population
Executes this particular implementation of selection.

This method is not called by the base algorithms implemented, but from `__call__()` instead. It should contain the logic of the specific selection schema.

Parameters

- **population** – The original population.
- **offspring** – The population to replace the original one.

Returns A new Population instance.

pynetics.selection module

Different implementations for some well known selection schemas.

class pynetics.selection.**ExponentialRank**(*, *alpha: float = 1*, *replacement: bool = False*)
Bases: *pynetics.selection.WeightedSelectionSchema*

Selection based on the fitness rank exponentially.

It's somewhat similar to the roulette wheel method but instead of assigning the probability to be selected proportionally to their fitness, their probability to be selected is exponentially to their position in a rank where the genotypes are ordered accordingly to their fitnesses.

If the beta parameter is 0, the schema is equivalent to the monte carlo one, i.e. every chromosome has the same probability to be chosen.

get_weights(*, *genotypes: Iterable[pynetics.api.Genotype]*) → Tuple[float, ...]
Weights will be arranged according to the genotypes position.

Because the fittest genotype is the last one, the weights will be in ascending order.

Parameters **genotypes** – The genotypes (only used to get the number of them).

Returns A tuple with as many frequencies as genotypes in the population.

class pynetics.selection.**LinearRank**(*, *alpha: float = 1*, *replacement: bool = False*)
Bases: *pynetics.selection.WeightedSelectionSchema*

Selection based on the fitness rank proportionally.

It's somewhat similar to the roulette wheel method but instead of assigning the probability to be selected proportionally to their fitness, their probability to be selected is proportional to their position in a rank where the genotypes are ordered accordingly to their fitnesses.

If the alpha parameter is 0, the schema is equivalent to the monte carlo one, i.e. every chromosome has the same probability to be chosen.

get_weights(*, *genotypes: Iterable[pynetics.api.Genotype]*) → Tuple[float, ...]
Weights will be arranged according to the genotypes position.

Because the fittest genotype is the last one, the weights will be in ascending order.

Parameters **genotypes** – The genotypes (only used to get the number of them).

Returns A tuple with as many frequencies as genotypes in the population.

class pynetics.selection.**MonteCarlo**(*replacement: bool = False*)
Bases: *pynetics.selection.SelectionSchema*

Selects the genotypes uniformly of the whole population.

It doesn't provide any means to increase the selective pressure across the generations. It's rather a way of measuring how well other selection schemes behave

do(*population: pynetics.api.Population*, *n: int*) → Iterable[pynetics.api.Genotype]
Implementation of the selection schema.

Parameters

- **population** – The population where genotypes are extracted.
- **n** – The number of genotypes to return.

Returns A sequence of genotypes.

class pynetics.selection.**RouletteWheel**(*replacement: bool = False*)
Bases: *pynetics.selection.WeightedSelectionSchema*

Selection by associating the fitness to a probabilities.

Also called "fitness proportionate selection", the function selects the genotypes weighting proportionally their fitness.

get_weights(*, *genotypes: Iterable[pynetics.api.Genotype]*) → Tuple[float, ...]
The frequency is directly each fitness.

Parameters **genotypes** – The genotypes to select from.

Returns A tuple with as many frequencies as genotypes in the population.

class pynetics.selection.**SelectionSchema**(*replacement: bool = False*)
Bases: object

Groups common behavior across all the selection schemas.

The selection schema is defined as a class. However, it is enough to implement it as a selection method, i.e. a function that receives a sequence and a number of individuals, and returns a sample of individuals of that size from the given population.

do(*population: pynetics.api.Population*, *n: int*) → Iterable[pynetics.api.Genotype]
Executes this particular implementation of selection.

This method is not called by the base algorithms implemented, but from `__call__()` instead. It should contain the logic of the specific selection schema.

Parameters

- **population** – The population where genotypes are extracted.
- **n** – The number of genotypes to return.

Returns A sequence of genotypes.

replacement
If selection schema is with or without replacement.

Returns True if is a selection with replacement or False otherwise.

class pynetics.selection.**Tournament**(*m: int*, *replacement: bool = False*)
Bases: *pynetics.selection.SelectionSchema*

Selects the best genotypes after random tournaments of genotypes.

The idea is as follows: If n genotypes are required, then n rounds (tournaments) of the following process are executed:

1. Select m genotypes uniformly across the whole population.
2. Select the best genotypes of those m genotypes.

After those n rounds are executed, the n resulting genotypes are returned.

As m is a meta-parameter of the genetic algorithm, it is necessary to create a new object with the desired value. For example, if a selection schema with m=3 is needed, the selection functor should be created as follows:

```
>>> tournament_m3 = Tournament(m=3)
```

After that, tournamet_m3 will be a tournament selection functor with an m of 3.

do(*population: pynetics.api.Population*, *n: int*) → Iterable[pynetics.api.Genotype]
Implementation of the selection schema.

Parameters

- **population** – The population where genotypes are extracted.
- **n** – The number of genotypes to return.

Returns A sequence of genotypes.

class pynetics.selection.**Truncation**(*replacement: bool = False*)
Bases: *pynetics.selection.SelectionSchema*

Selects the best genotypes after sorting the population.

This selector simply selects the best n individuals from the population.

do(*population: pynetics.api.Population*, *n: int*) → Iterable[pynetics.api.Genotype]
Implementation of the selection schema.

Parameters

- **population** – The population where genotypes are extracted.
- **n** – The number of genotypes to return.

Returns A sequence of genotypes.

class pynetics.selection.**WeightedSelectionSchema**(*replacement: bool = False*)
Bases: *pynetics.selection.SelectionSchema*

Generic behaviour to all the weight based schemas.

do(*population: pynetics.api.Population*, *n: int*) → Iterable[pynetics.api.Genotype]
Implementation of the selection schema.

Parameters

- **population** – The population where genotypes are extracted.
- **n** – The number of genotypes to return.

Returns A sequence of genotypes.

get_weights(**, genotypes: Iterable[pynetics.api.Genotype]*) → Tuple[float, ...]
Computes as many probabilities as genotypes.

The probabilities depend completely of the implementing class.

Parameters **genotypes** – The genotypes to select from.

Returns A tuple with as many frequencies as genotypes in the population.

pynetics.stop module

Different implementations of stop conditions for algorithms.

class pynetics.stop.**FitnessBound**(*bound: float*)
Bases: object

Stop based on a fitness bound.

class pynetics.stop.**NumSteps**(*steps: int*)
Bases: object

Stop based on the number of iterations.

The condition is met once the number of iterations has reached an specified limit.

pynetics.util module

Utilities to be used across the project.

pynetics.util.**take_chances**(*probability: float = 0.5*) → bool
Random bool value given a probability p.

Parameters **probability** – The value of the probability to beat. Defaults to 0.5.

Returns A true value with a probability of p (thus, a False value with a (1 - p) probability).

Module contents

Import core names of pynetics.

This library provides the means to create genetic algorithms in a simple yet fast way. Specifically:

1. A generic and modular algorithm to be composed by the different parts that build a whole algorithm. 2. Simple algorithm implementations with some defaults implemented. 3. Different generic operators (e.g. crossover or mutation) to work with different genotypes (e.g. binary list, real list).

The fastest way to work with this library is by importing this file directly:

```
>>> import pynetics as pyn
```

TODO Organize in a way that this actually happens

CHAPTER 6

Backmatter

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

W