Table of Contents

Previous topic
Evolutionary Tools

Next topic
Genetic Programming

This Page
Show Source

Quick search
Go

v: master

Algorithms

The `algorithms` module is intended to contain some specific algorithms in order to execute very common evolutionary algorithms. The method used here are more for convenience than reference as the implementation of every evolutionary algorithm may vary infinitely. Most of the algorithms in this module use operators registered in the toolbox. Generally, the keyword used are `mate()` for crossover, `mutate()` for mutation, `select()` for selection and `evaluate()` for evaluation.

You are encouraged to write your own algorithms in order to make them do what you really want them to do.

Complete Algorithms

These are complete boxed algorithms that are somewhat limited to the very basic evolutionary computation concepts. All algorithms accept, in addition to their arguments, an initialized `Statistics` object to maintain stats of the evolution, an initialized `HallOfFame` to hold the best individual(s) to appear in the population, and a boolean *verbose* to specify whether to log what is happening during the evolution or not.

`deap.algorithms.`**`eaSimple`**(*population*, *toolbox*, *cxpb*, *mutpb*, *ngen*[, *stats*, *halloffame*, *verbose*])

This algorithm reproduce the simplest evolutionary algorithm as presented in chapter 7 of [Back2000].

Parameters:	• **population** – A list of individuals. • **toolbox** – A `Toolbox` that contains the evolution operators. • **cxpb** – The probability of mating two individuals. • **mutpb** – The probability of mutating an individual. • **ngen** – The number of generation. • **stats** – A `Statistics` object that is updated inplace, optional. • **halloffame** – A `HallOfFame` object that will contain the best individuals, optional. • **verbose** – Whether or not to log the statistics.
Returns:	The final population
Returns:	A class:~*deap.tools.Logbook* with the statistics of the evolution

The algorithm takes in a population and evolves it in place using the `varAnd()` method. It returns the optimized population and a `Logbook` with the statistics of the evolution. The logbook will contain the generation number, the number of evalutions for each generation and the statistics if a `Statistics` is given as argument. The *cxpb* and *mutpb* arguments are passed to the `varAnd()` function. The pseudocode goes as follow

```
evaluate(population)
for g in range(ngen):
    population = select(population, len(population))
    offspring = varAnd(population, toolbox, cxpb, mutpb)
    evaluate(offspring)
    population = offspring
```

As stated in the pseudocode above, the algorithm goes as follow. First, it evaluates the individuals with an invalid fitness. Second, it enters the generational loop where the selection procedure is applied to entirely replace the parental population. The 1:1 replacement ratio of this algorithm **requires** the selection procedure to be stochastic and to select multiple times the same individual, for example, `selTournament()` and `selRoulette()`. Third, it applies the `varAnd()` function to produce the next generation population. Fourth, it evaluates the new individuals and compute the statistics on this population. Finally, when *ngen* generations are done, the algorithm returns a tuple with the final population and a `Logbook` of the evolution.

Note: Using a non-stochastic selection method will result in no selection as the operator selects *n* individuals from a pool of *n*.

This function expects the `toolbox.mate()`, `toolbox.mutate()`, `toolbox.select()` and `toolbox.evaluate()` aliases to be registered in the toolbox.

[Back2000] Back, Fogel and Michalewicz, “Evolutionary Computation 1 : Basic Algorithms and Operators”, 2000.

`deap.algorithms.`**`eaMuPlusLambda`**(*population*, *toolbox*, *mu*, *lambda_*, *cxpb*, *mutpb*, *ngen*[, *stats*, *halloffame*, *verbose*])

This is the $(\mu + \lambda)$ evolutionary algorithm.

Parameters:	• **population** – A list of individuals. • **toolbox** – A `Toolbox` that contains the evolution operators. • **mu** – The number of individuals to select for the next generation. • **lambda_** – The number of children to produce at each generation. • **cxpb** – The probability that an offspring is produced by crossover. • **mutpb** – The probability that an offspring is produced by mutation. • **ngen** – The number of generation. • **stats** – A `Statistics` object that is updated inplace, optional. • **halloffame** – A `HallOfFame` object that will contain the best individuals, optional. • **verbose** – Whether or not to log the statistics.
Returns:	The final population
Returns:	A class:~*deap.tools.Logbook* with the statistics of the evolution.

The algorithm takes in a population and evolves it in place using the `varOr()` function. It returns the optimized population and a `Logbook` with the statistics of the evolution. The logbook will contain the generation number, the number of evalutions for each generation and the statistics if a `Statistics` is given as argument. The *cxpb* and *mutpb* arguments are passed to the `varOr()` function. The pseudocode goes as follow

```
evaluate(population)
for g in range(ngen):
    offspring = varOr(population, toolbox, lambda_, cxpb, mutpb)
    evaluate(offspring)
    population = select(population + offspring, mu)
```

First, the individuals having an invalid fitness are evaluated. Second, the evolutionary loop begins by producing *lambda_* offspring from the population, the offspring are generated by the `varOr()` function. The offspring are then evaluated and the next generation population is selected from both the offspring **and** the population. Finally, when *ngen* generations are done, the algorithm returns a tuple with the final population and a `Logbook` of the evolution.

This function expects `toolbox.mate()`, `toolbox.mutate()`, `toolbox.select()` and `toolbox.evaluate()` aliases to be registered in the toolbox. This algorithm uses the `varOr()` variation.

`deap.algorithms.`**`eaMuCommaLambda`**(*population*, *toolbox*, *mu*, *lambda_*, *cxpb*, *mutpb*, *ngen*[, *stats*, *halloffame*, *verbose*])

This is the $(\mu~,~\lambda)$ evolutionary algorithm.

Parameters:	• **population** – A list of individuals. • **toolbox** – A `Toolbox` that contains the evolution operators. • **mu** – The number of individuals to select for the next generation. • **lambda_** – The number of children to produce at each generation. • **cxpb** – The probability that an offspring is produced by crossover. • **mutpb** – The probability that an offspring is produced by mutation. • **ngen** – The number of generation. • **stats** – A `Statistics` object that is updated inplace, optional. • **halloffame** – A `HallOfFame` object that will contain the best individuals, optional. • **verbose** – Whether or not to log the statistics.
Returns:	The final population
Returns:	A class:~*deap.tools.Logbook* with the statistics of the evolution

The algorithm takes in a population and evolves it in place using the `varOr()` function. It returns the optimized population and a `Logbook` with the statistics of the evolution. The logbook will contain the generation number, the number of evalutions for each generation and the statistics if a `Statistics` is given as argument. The *cxpb* and *mutpb* arguments are passed to the `varOr()` function. The pseudocode goes as follow

```
evaluate(population)
for g in range(ngen):
    offspring = varOr(population, toolbox, lambda_, cxpb, mutpb)
    evaluate(offspring)
    population = select(offspring, mu)
```

First, the individuals having an invalid fitness are evaluated. Second, the evolutionary loop begins by producing *lambda_* offspring from the population, the offspring are generated by the `varOr()` function. The offspring are then evaluated and the next generation population is selected from **only** the offspring. Finally, when *ngen* generations are done, the algorithm returns a tuple with the final population and a `Logbook` of the evolution.

Note: Care must be taken when the lambda:mu ratio is 1 to 1 as a non-stochastic selection will result in no selection at all as the operator selects *lambda* individuals from a pool of *mu*.

This function expects `toolbox.mate()`, `toolbox.mutate()`, `toolbox.select()` and `toolbox.evaluate()` aliases to be registered in the toolbox. This algorithm uses the `varOr()` variation.

`deap.algorithms.`**`eaGenerateUpdate`**(*toolbox*, *ngen*[, *stats*, *halloffame*, *verbose*])

This is algorithm implements the ask-tell model proposed in [Colette2010], where ask is called *generate* and tell is called *update*.

Parameters:	• **toolbox** – A `Toolbox` that contains the evolution operators. • **ngen** – The number of generation. • **stats** – A `Statistics` object that is updated inplace, optional. • **halloffame** – A `HallOfFame` object that will contain the best individuals, optional. • **verbose** – Whether or not to log the statistics.
Returns:	The final population
Returns:	A class:~*deap.tools.Logbook* with the statistics of the evolution

The algorithm generates the individuals using the `toolbox.generate()` function and updates the generation method with the `toolbox.update()` function. It returns the optimized population and a `Logbook` with the statistics of the evolution. The logbook will contain the generation number, the number of evalutions for each generation and the statistics if a `Statistics` is given as argument. The pseudocode goes as follow

```
for g in range(ngen):
    population = toolbox.generate()
    evaluate(population)
    toolbox.update(population)
```

[Colette2010] Collette, Y., N. Hansen, G. Pujol, D. Salazar Aponte and R. Le Riche (2010). On Object-Oriented Programming of Optimizers - Examples in Scilab. In P. Breitkopf and R. F. Coelho, eds.: Multidisciplinary Design Optimization in Computational Mechanics, Wiley, pp. 527-565;

Variations

Variations are smaller parts of the algorithms that can be used separately to build more complex algorithms.

`deap.algorithms.`**`varAnd`**(*population*, *toolbox*, *cxpb*, *mutpb*)

Part of an evolutionary algorithm applying only the variation part (crossover **and** mutation). The modified individuals have their fitness invalidated. The individuals are cloned so returned population is independent of the input population.

Parameters:	• **population** – A list of individuals to vary. • **toolbox** – A `Toolbox` that contains the evolution operators. • **cxpb** – The probability of mating two individuals. • **mutpb** – The probability of mutating an individual.
Returns:	A list of varied individuals that are independent of their parents.

The variation goes as follow. First, the parental population P_p is duplicated using the `toolbox.clone()` method and the result is put into the offspring population P_o. A first loop over P_o is executed to mate pairs of consecutive individuals. According to the crossover probability *cxpb*, the individuals $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ are mated using the `toolbox.mate()` method. The resulting children $\mathbf{y}_i$ and $\mathbf{y}_{i+1}$ replace their respective parents in P_o. A second loop over the resulting P_o is executed to mutate every individual with a probability *mutpb*. When an individual is mutated it replaces its not mutated version in P_o. The resulting P_o is returned.

This variation is named *And* beceause of its propention to apply both crossover and mutation on the individuals. Note that both operators are not applied systematicaly, the resulting individuals can be generated from crossover only, mutation only, crossover and mutation, and reproduction according to the given probabilities. Both probabilities should be in $[0, 1]$.

`deap.algorithms.`**`varOr`**(*population*, *toolbox*, *lambda_*, *cxpb*, *mutpb*)

Part of an evolutionary algorithm applying only the variation part (crossover, mutation **or** reproduction). The modified individuals have their fitness invalidated. The individuals are cloned so returned population is independent of the input population.

Parameters:	• **population** – A list of individuals to vary. • **toolbox** – A `Toolbox` that contains the evolution operators. • **lambda_** – The number of children to produce • **cxpb** – The probability of mating two individuals. • **mutpb** – The probability of mutating an individual.
Returns:	The final population.

The variation goes as follow. On each of the *lambda_* iteration, it selects one of the three operations; crossover, mutation or reproduction. In the case of a crossover, two individuals are selected at random from the parental population P_p, those individuals are cloned using the `toolbox.clone()` method and then mated using the `toolbox.mate()` method. Only the first child is appended to the offspring population P_o, the second child is discarded. In the case of a mutation, one individual is selected at random from P_p, it is cloned and then mutated using using the `toolbox.mutate()` method. The resulting mutant is appended to P_o. In the case of a reproduction, one individual is selected at random from P_p, cloned and appended to P_o.

This variation is named *Or* beceause an offspring will never result from both operations crossover and mutation. The sum of both probabilities shall be in $[0, 1]$, the reproduction probability is 1 - *cxpb* - *mutpb*.

Covariance Matrix Adaptation Evolution Strategy

A module that provides support for the Covariance Matrix Adaptation Evolution Strategy.

class `deap.cma.`**`Strategy`**(*centroid*, *sigma*[, ***kargs*])

A strategy that will keep track of the basic parameters of the CMA-ES algorithm ([Hansen2001]).

Parameters:	• **centroid** – An iterable object that indicates where to start the evolution. • **sigma** – The initial standard deviation of the distribution. • **parameter** – One or more parameter to pass to the strategy as described in the following table, optional.

Parameter	Default	Details
`lambda_`	`int(4 + 3 * log(N))`	Number of children to produce at each generation, `N` is the individual’s size (integer).
`mu`	`int(lambda_ / 2)`	The number of parents to keep from the lambda children (integer).
`cmatrix`	`identity(N)`	The initial covariance matrix of the distribution that will be sampled.
`weights`	`"superlinear"`	Decrease speed, can be `"superlinear"`, `"linear"` or `"equal"`.
`cs`	`(mueff + 2) / (N + mueff + 3)`	Cumulation constant for step-size.
`damps`	`1 + 2 * max(0, sqrt(( mueff - 1) / (N + 1)) - 1) + cs`	Damping for step-size.
`ccum`	`4 / (N + 4)`	Cumulation constant for covariance matrix.
`ccov1`	`2 / ((N + 1.3)^2 + mueff)`	Learning rate for rank-one update.
`ccovmu`	`2 * (mueff - 2 + 1 / mueff) / ((N + 2)^2 + mueff)`	Learning rate for rank-mu update.

[Hansen2001] Hansen and Ostermeier, 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*

`computeParams`(*params*)

Computes the parameters depending on λ. It needs to be called again if λ changes during evolution.

Parameters:	**params** – A dictionary of the manually set parameters.

`generate`(*ind_init*)

Generate a population of λ individuals of type *ind_init* from the current strategy.

Parameters:	**ind_init** – A function object that is able to initialize an individual from a list.
Returns:	A list of individuals.

`update`(*population*)

Update the current covariance matrix strategy from the *population*.

Parameters:	**population** – A list of individuals from which to update the parameters.

class `deap.cma.`**`StrategyOnePlusLambda`**(*parent*, *sigma*[, ***kargs*])

A CMA-ES strategy that uses the $1 + \lambda$ paradigm ([Igel2007]).

Parameters:	• **parent** – An iterable object that indicates where to start the evolution. The parent requires a fitness attribute. • **sigma** – The initial standard deviation of the distribution. • **lambda** – Number of offspring to produce from the parent. (optional, defaults to 1) • **parameter** – One or more parameter to pass to the strategy as described in the following table. (optional)

Other parameters can be provided as described in the next table

Parameter	Default	Details
`d`	`1.0 + N / (2.0 * lambda_)`	Damping for step-size.
`ptarg`	`1.0 / (5 + sqrt(lambda_) / 2.0)`	Taget success rate.
`cp`	`ptarg * lambda_ / (2.0 + ptarg * lambda_)`	Step size learning rate.
`cc`	`2.0 / (N + 2.0)`	Cumulation time horizon.
`ccov`	`2.0 / (N**2 + 6.0)`	Covariance matrix learning rate.
`pthresh`	`0.44`	Threshold success rate.

[Igel2007] Igel, Hansen, Roth, 2007. Covariance matrix adaptation for

multi-objective optimization. *Evolutionary Computation* Spring;15(1):1-28

`computeParams`(*params*)

Computes the parameters depending on λ. It needs to be called again if λ changes during evolution.

Parameters:	**params** – A dictionary of the manually set parameters.

`generate`(*ind_init*)

Generate a population of λ individuals of type *ind_init* from the current strategy.

Parameters:	**ind_init** – A function object that is able to initialize an individual from a list.
Returns:	A list of individuals.

`update`(*population*)

Update the current covariance matrix strategy from the *population*.

Parameters:	**population** – A list of individuals from which to update the parameters.

class `deap.cma.`**`StrategyMultiObjective`**(*population*, *sigma*[, ***kargs*])

Multiobjective CMA-ES strategy based on the paper [Voss2010]. It is used similarly as the standard CMA-ES strategy with a generate-update scheme.

Parameters:	• **population** – An initial population of individual. • **sigma** – The initial step size of the complete system. • **mu** – The number of parents to use in the evolution. When not provided it defaults to the length of *population*. (optional) • **lambda** – The number of offspring to produce at each generation. (optional, defaults to 1) • **indicator** – The indicator function to use. (optional, default to `hypervolume()`)

Other parameters can be provided as described in the next table

Parameter	Default	Details
`d`	`1.0 + N / 2.0`	Damping for step-size.
`ptarg`	`1.0 / (5 + 1.0 / 2.0)`	Taget success rate.
`cp`	`ptarg / (2.0 + ptarg)`	Step size learning rate.
`cc`	`2.0 / (N + 2.0)`	Cumulation time horizon.
`ccov`	`2.0 / (N**2 + 6.0)`	Covariance matrix learning rate.
`pthresh`	`0.44`	Threshold success rate.

[Voss2010] Voss, Hansen, Igel, “Improved Step Size Adaptation for the MO-CMA-ES”, 2010.

`generate`(*ind_init*)

Generate a population of λ individuals of type *ind_init* from the current strategy.

Parameters:	**ind_init** – A function object that is able to initialize an individual from a list.
Returns:	A list of individuals with a private attribute `_ps`. This last attribute is essential to the update function, it indicates that the individual is an offspring and the index of its parent.

`update`(*population*)

Update the current covariance matrix strategies from the *population*.

Parameters:	**population** – A list of individuals from which to update the parameters.