TP 3 – Métaheuristique

Les algorithmes génétiques

Bellanger Clément

Résultats

Les tests on été effectué sur les deux problèmes avec des solutions binaires: Sac à dos et set covering

Pour les test, on compare RandomLS à MuComaLambdaGA, MuPlusComaLambdaGA et OnePlusOneEA

Les tests ont étés effectués avec les paramètres suivants :

Pour chaque recherche par population :

alpha : 0,1

mu : 10

lambda : 20

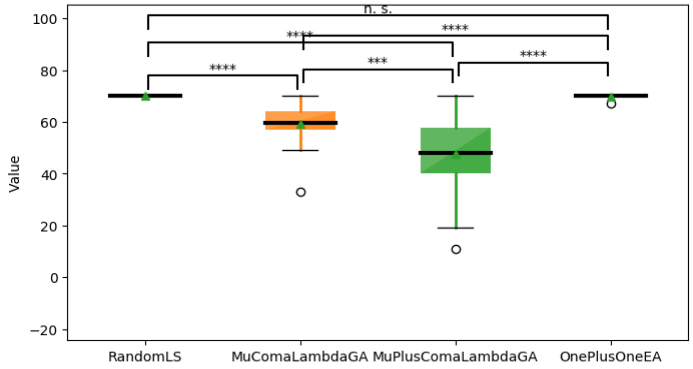
MuComaLambdaGA et MuPlusComaLambdaGA

probabilité de croisement : 0,2

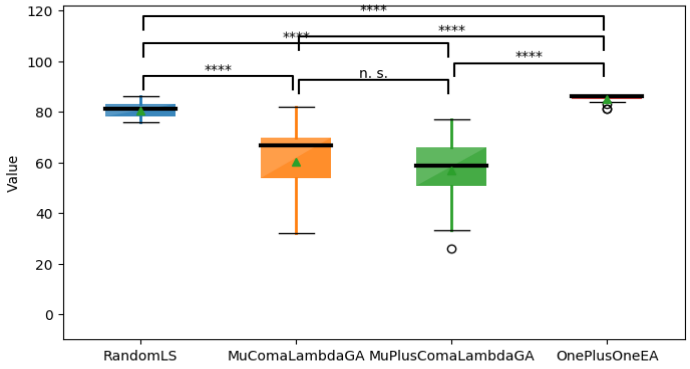
probabilité de mutation : 0,2

Problème – Sac à Dos

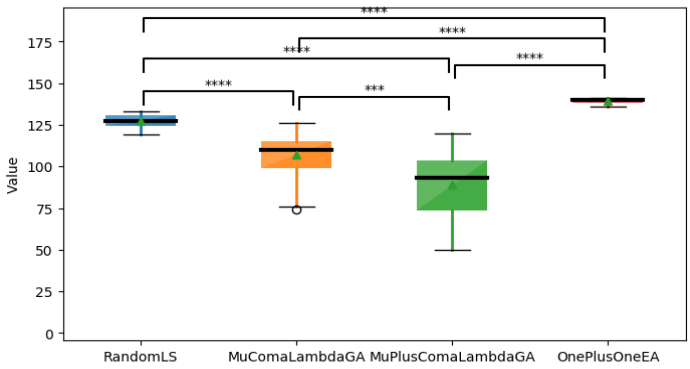
Taille – small



Taille – medium



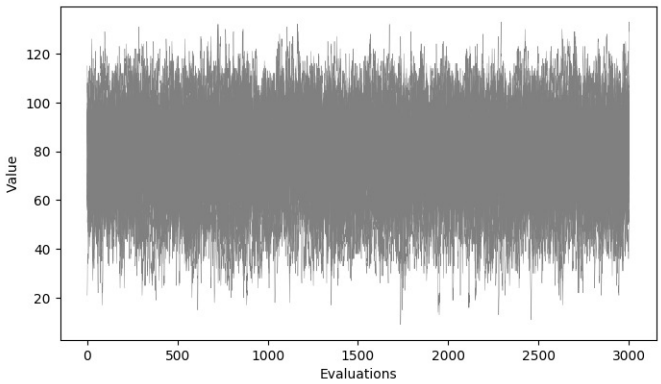
Taille – large



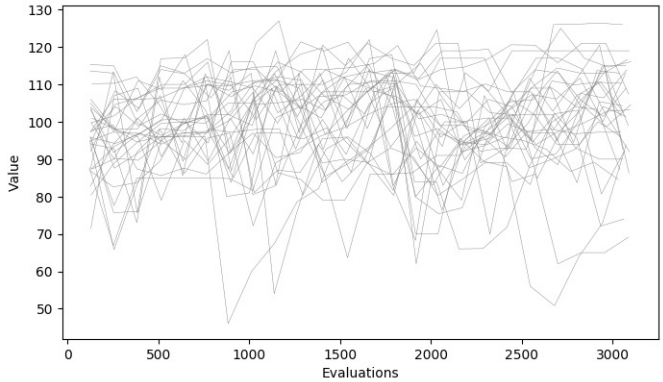
Problème – Sac à Dos

Trace des algorithmes (taille large)

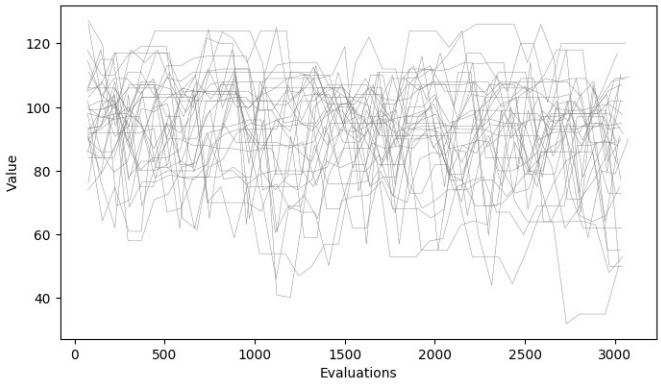
RandomLS



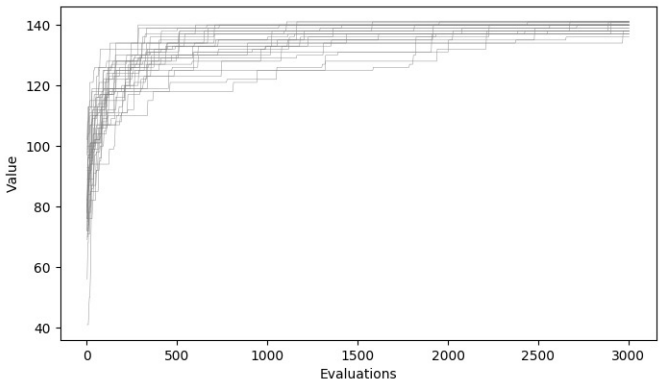
MuComaLambdaGA



MuPlusComaLambdaGA

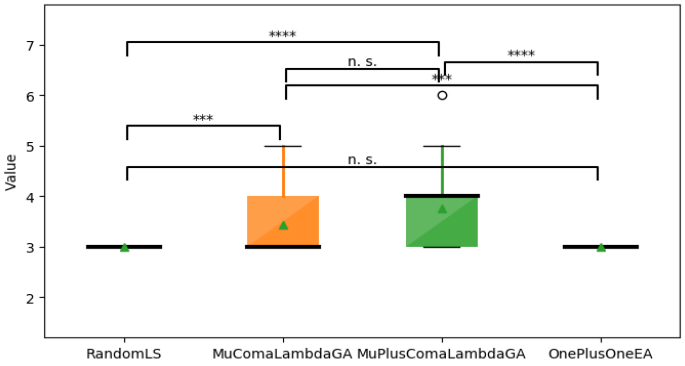


OnePlusOneEA

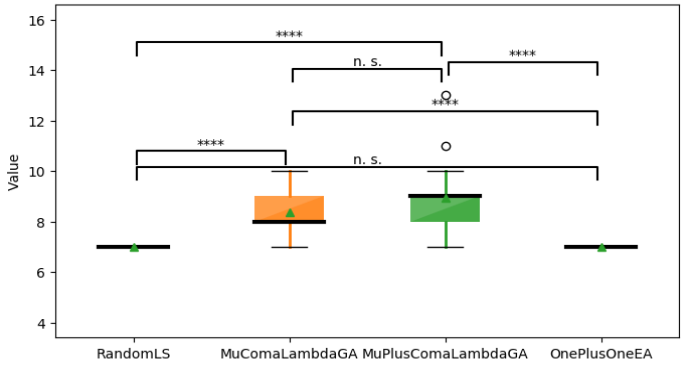


Problème – Set Covering

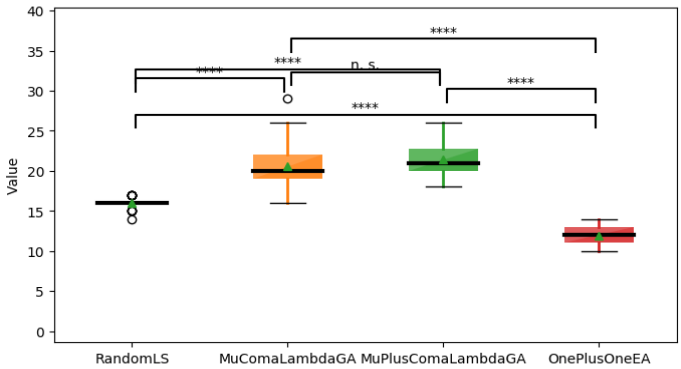
Taille – small



Taille – medium



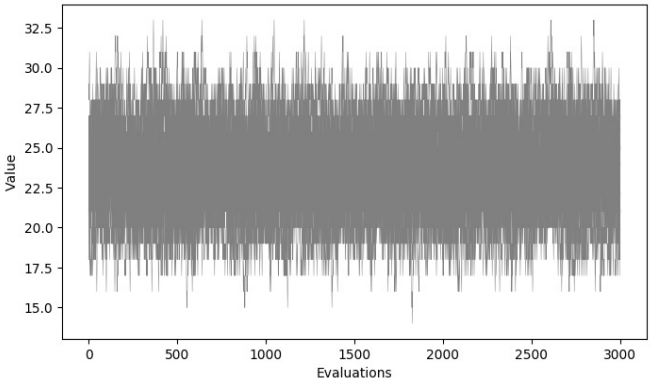
Taille – large



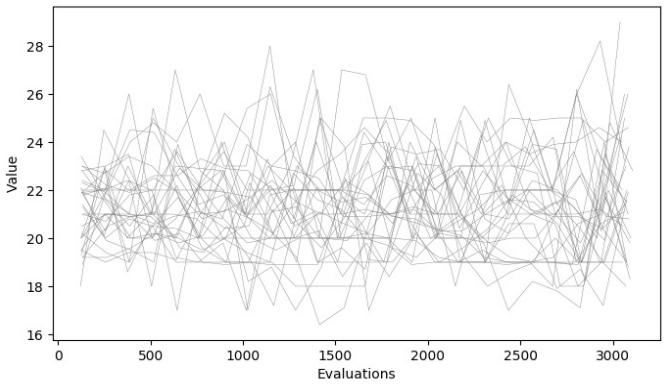
Problème – Set Covering

Trace des algorithmes (taille large)

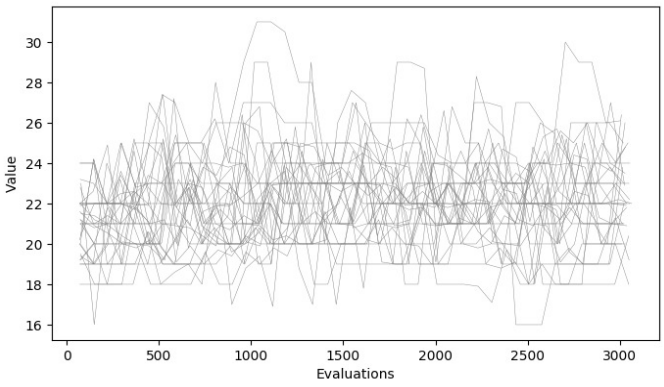
RandomLS



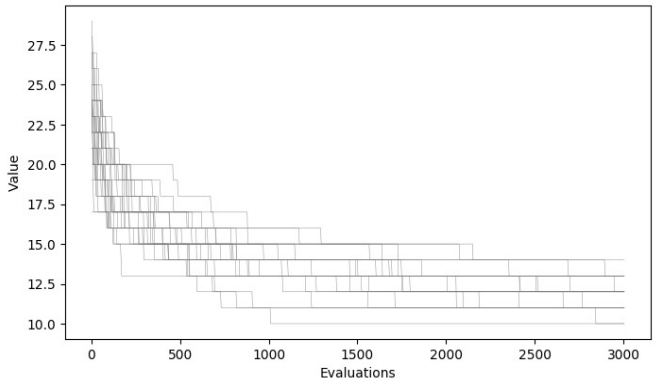
MuComaLambdaGA



MuPlusComaLambdaGA



OnePlusOneEA



Commentaire

On constate que l’algorithme génétique frustré OnePlusOneEA réussit le mieux pour ces deux types de problème.

En modifiant la valeur alpha, on constate une chose, plus la probabilité est grande plus elle dégrade les solutions lors de la mutation uniforme et plus l’algorithme a un comportement exploratoire.

On a donc laissé cette probabilité volontairement basse pour avoir des mutations légères seulement.

Suite à ces résultat, on peut se demander si le croisement par point est une bonne solution.

Le problème est que avec les algorithmes actuels pour le problème du sac à dos, le croisement uniforme met beaucoup de temps à générer deux enfants valides et donc ne peut pas être utilisé en pratique.

Code commenté

```python
#! /usr/bin/env python
# -*- coding: utf-8 -*-
import numpy as np

import random

from generic import *

# Structure temporaire à améliorer
alpha=0.1

class OnePlusOneEA(PopulationSearchAlgorithm):

    def __init__ (self, prob, options) :
        """
        Entrées :

        * Un problème donnée instance de la classe Problem.
        Seules les problèmes dont les solution sont des tableaus booléens sont
        acceptée.

        * Un dictionnaire des paramètres des algorithmes

        """

        # La taille de la popilation est toujours 1 pour cet algorithme
        options['mu']=1
        options['lambda']=1

        PopulationSearchAlgorithm.__init__(self, prob, options)


    def make_parent_pop(self):
        """
        Permet de selectionner selon une strategie donnée, qui parmi la
        population courrante (self._pop) va pouvoir se reproduire.

        retourne une liste de solution de self._pop de taille self._mu

        """
        # Population de mu parents choisie aléatoirement de self._pop
        parents = []
        for _ in xrange( len(self._pop) ):
            parents.append( random.choice(self._pop) )
        return parents


    def select_random(self, pop):
        """
        On selectionnne une solution aleatoire de la population donnée

        Entrée : une liste de solutions
        Sortie : une solution (aléatoire)

        """

        return random.choice( pop ).clone()


    def bit_uniform_mutation(self, x) :
        """
        Mutation d'un individu. On modifie uniformément chaque bit avec une probabilité alpha.

        Entrée : une instance de la classe Solution.

        Sortie : la solution modifiée
        """
        if not isinstance(x, BinarySolution) :
            raise TypeError("Algorithm only works on binary solution problems")

        for i in xrange(len(x.solution)):
            if (alpha > random.uniform(0, 1)) :
```

```
                x.solution[i] = not x.solution[i]

        return x


    def evolve_pop(self, parents):
        """
        Créer des nouvelles solution par evolution des parents. Les opérateur
        génétique son appliqué ici.

        Entrée : une liste de parents (cf.  make_parent_pop)
        Sortie : une liste de solution enfants de taille self._lambda

        """
        offspring = []
        done = False
        while not done :

            # on prend une solution
            x = self.select_random ( parents )
            self._problem.eval(x)

            # on la modifie avec une mutation uniforme et on la garde seulement si elle est meilleure
            y = x.clone()
            y = self.bit_uniform_mutation( y )
            self._problem.eval(y)

            # on a rajoute a la liste la meilleure solution si elle est faisable
            if self.better(y.value,x.value) and self._problem.feasable(y):
                offspring.append(y)
            else:
                offspring.append(x)


            # on arrête si la population est remplie
            done = len(offspring) == self._lambda

        return offspring

    def make_new_pop(self, offspring):
        """
        Constituer la nouvelle population selon une stratégie de selection
        donnée depuis les enfants et self._pop.

        Entrée : une liste de solutions enfants
        Sortie : une liste de solutions de taille self._mu

        """

        # On tri les enfants
        self.sort_pop(offspring)

        # on garde les mu meilleurs
        survivors = []
        for i in xrange( self._mu ):
            survivors.append( offspring[i] )
        return survivors
```

```python
#! /usr/bin/env python
# -*- coding: utf-8 -*-
import numpy as np
import copy
import random

from generic import *

# Structure temporaire à améliorer
alpha=0.1

# Probabilité de croisement
pc=0.2

# Probabilité de mutation
pm=0.2
class MuComaLambdaGA(PopulationSearchAlgorithm):

    def __init__ (self, prob, options) :
        """
        Entrées :

        * Un problème donnée instance de la classe Problem.
        Seules les problèmes dont les solution sont des tableaus booléens sont
        acceptée.

        * Un dictionnaire des paramètres des algorithmes

        """
        PopulationSearchAlgorithm.__init__(self, prob, options)


    def make_parent_pop(self):
        """
        Permet de selectionner selon une strategie donnée, qui parmi la
        population courrante (self._pop) va pouvoir se reproduire.

        retourne une liste de solution de self._pop de taille self._mu

        """
        # Population de mu parents choisie aléatoirement de self._pop
        parents = []
        for _ in xrange( len(self._pop) ):
            parents.append( random.choice(self._pop) )
        return parents


    def select_random(self, pop):
        """
        On selectionnne une solution aleatoire de la population donnée

        Entrée : une liste de solutions
        Sortie : une solution (aléatoire)

        """

        return random.choice( pop ).clone()

    def select2_random(self, pop):
        """
        On selectionnne deux solutions aleatoire différentes de la population donnée

        Entrée : une liste de solutions
        Sortie : une liste de deux solutions (aléatoire)

        """
        twosolutions = []

        # Si la population contient au moins deux solutions
        if len(pop) >= 2:

            solution=pop[0]
```

```
            twodifferent=False

            # On vérifie qu'il existe deux solutions différentes
            for i in xrange(1,len(pop)):
                 if pop[i] != solution:
                     twodifferent=True
                     break

            # Si toutes les solutions sont identiques, on retourne les deux premières
            if twodifferent == False:
                twosolutions.append(pop[0])
                twosolutions.append(pop[1])
                return twosolutions

            # On sélectionne au hasard une solution x et y
            x = random.choice( pop ).clone()
            y = random.choice( pop ).clone()

            # Tant que l'on a les memes solution
            while y == x:
                y = random.choice( pop ).clone()
            twosolutions.append(x)
            twosolutions.append(y)

            return twosolutions

        else:
            print("Erreur la population ne possède pas deux solutions")
            return twosolutions

    def bit_uniform_mutation(self, x) :
        """
        Mutation d'un individu. On modifie uniformément chaque bit avec une probabilité alpha.

        Entrée : une instance de la classe Solution.

        Sortie : la solution modifiée
        """
        if not isinstance(x, BinarySolution) :
            raise TypeError("Algorithm only works on binary solution problems")

        for i in xrange(len(x.solution)):
            if (alpha > random.uniform(0, 1)) :
                x.solution[i] = not x.solution[i]

        return x

    def bit_uniform_crossing(self, x, y):
        """
        Croisement uniforme de deux individus. On a pour chaque gène la probabilité 1/2 d'avoir le
gène du parent 1 et 1/2 d'avoir celle du parent 2

        Entrée : deux instances de la classe Solution.

        Sortie : les deux solutions enfants
        """
        if not isinstance(x, BinarySolution) or not isinstance(y, BinarySolution) :
            raise TypeError("Algorithm only works on binary solution problems")

        # On crée une copie de chaque solution
        xprime = x.clone()
        yprime = y.clone()


        # Pour chaque bit on 1/2 d'avoir le bit du parent 1 ou du parent 2

        # Enfant 1
        for i in xrange(len(x.solution)):
            if (0.5 > random.uniform(0, 1)) :
                xprime.solution[i] = x.solution[i]
            else :
                xprime.solution[i] = y.solution[i]
```

```
        # Enfant 2
        for i in xrange(len(x.solution)):
            if (0.5 > random.uniform(0, 1)) :
                yprime.solution[i] = x.solution[i]
            else :
                yprime.solution[i] = y.solution[i]

        # On retourne une liste avec chaque solution enfant
        listchild = []
        listchild.append(xprime)
        listchild.append(yprime)

        return listchild


    def bit_point_crossing(self, x, y):
        """
        Croisement en un point de deux individus. On sélectionne un indice. Avant celui-ci l'enfant 1
aura tous les gènes du parent 1 et après
        celui-ci il aura tous les gènes du parent 2. Inversement pour l'enfant 2.

        Entrée : deux instances de la classe Solution.

        Sortie : les deux solutions enfants
        """
        if not isinstance(x, BinarySolution) or not isinstance(y, BinarySolution) :
            raise TypeError("Algorithm only works on binary solution problems")

        xprime = x.clone()
        yprime = y.clone()

        randomindice=np.random.randint(len(x.solution))

        for i in xrange(len(x.solution)):
            if i < randomindice:
                # Enfant 1
                xprime.solution[i] = x.solution[i]
                # Enfant 2
                yprime.solution[i] = y.solution[i]
            else:
                # Enfant 1
                xprime.solution[i] = y.solution[i]
                # Enfant 2
                yprime.solution[i] = x.solution[i]


        listchild = []
        listchild.append(xprime)
        listchild.append(yprime)

        return listchild

    def tournament_selection(self, pop, n, k):
        """
        Sélection par tournoi de n individidu. On prend n fois le meilleur des échantillons de k

        Entrée : Une population (liste) de solutions, le nombre n de solutions à sélectionner, le
nombre k de solutions par échantillion

        Sortie : La population des n sélectionné
        """

        # La population finale des solutions gagnante de chaque tournoi
        npop = []

        # On crée un clone de la poupulation pour ne pas la modifier
        popClone=copy.copy(pop)

        # On sélectionne n individu
        for j in xrange(n):
```

```
            # On crée une copie de la population de sorte à ce que les solutions puissent faire
plusieurs tournois (sauf les gagnants, chaque tournoi est sans remise)
            popCopy=copy.copy(popClone)

            # La population (échantillon) des k individus sur lesquels la sélection va s'opérer
            kpop = []

            # On sélectionne k individu sans remise
            for i in xrange(k):
                randomindice=np.random.randint(len(popCopy))
                ksolution=popCopy.pop(randomindice)
                kpop.append(ksolution)

            best_solution=kpop[0]
            best_evaluation=self._problem.eval(best_solution)
            for i in xrange(1,len(kpop)):

                # On évalue les solutions parcourues
                current_evaluation = self._problem.eval(kpop[i])

                # Si la solution est meilleure que la solution courante, on sauvegarde la solution
(best_evaluation) et sa valeur (best_evaluation)
                if self.better(current_evaluation,best_evaluation):
                    best_solution = kpop[i]
                    best_evaluation = current_evaluation

            # La solution a gagné le tournoi, elle est donc sélectionné pour etre dans la population
finale
            npop.append(best_solution)

            # On supprime la solution de la population initiale
            popClone.remove(best_solution)

        return npop

    def evolve_pop(self, parents):
        """
        Créer des nouvelles solution par evolution des parents. Les opérateur
        génétique son appliqué ici.

        Entrée : une liste de parents (cf.  make_parent_pop)
        Sortie : une liste de solution enfants de taille self._lambda

        """
        offspring = []
        done = False
        while not done :

            # On sélectionne les géniteurs
            y,z = self.tournament_selection(parents,2,5)

            # On a une certaine probabilité de croisement
            if (pc > random.uniform(0, 1)):
                feasable=False
                # On cherche a obtenir des solutions enfants qui sont valides
                while (not feasable):
                    childs=self.bit_point_crossing(y,z)
                    y=childs[0]
                    z=childs[1]
                    feasable=self._problem.feasable(y) and self._problem.feasable(z)
                self._problem.eval(y)
                self._problem.eval(z)

            # On a une certaine probabilité de mutation
            # On cherche a obtenir des mutations qui sont valides
            if (pm > random.uniform(0, 1)):
                feasable=False
                while (not feasable):
                    y=self.bit_uniform_mutation(y)
                    feasable=self._problem.feasable(y)
                feasable=False
                while ( not feasable):
```

```
                z=self.bit_uniform_mutation(z)
                feasable=self._problem.feasable(z)
            self._problem.eval(y)
            self._problem.eval(z)

        # on a rajoute a la liste
        offspring.append(y)
        offspring.append(z)

        # on arrête si la population est remplie
        done = len(offspring) == self._lambda

    return offspring

def make_new_pop(self, offspring):
    """
    Constituer la nouvelle population selon une stratégie de selection
    donnée depuis les enfants et self._pop.

    Entrée : une liste de solutions enfants
    Sortie : une liste de solutions de taille self._mu

    """

    # On tri les enfants
    self.sort_pop(offspring)

    # on garde les mu meilleurs
    survivors = []
    for i in xrange( self._mu ):
        survivors.append( offspring[i] )
    return survivors
```

```python
#! /usr/bin/env python
# -*- coding: utf-8 -*-
import numpy as np
import copy
import random

from mu_coma_lambda_ga import MuComaLambdaGA
from generic import *

# Structure temporaire à améliorer
alpha=0.1

# Probabilité de croisement
pc=0.2

# Probabilité de mutation
pm=0.2
class MuPlusComaLambdaGA(MuComaLambdaGA):

    def __init__ (self, prob, options) :
        """
        Entrées :

        * Un problème donnée instance de la classe Problem.
        Seules les problèmes dont les solution sont des tableaus booléens sont
        acceptée.

        * Un dictionnaire des paramètres des algorithmes

        """
        MuComaLambdaGA.__init__(self, prob, options)

    def evolve_pop(self, parents):
        """
        Créer des nouvelles solution par evolution des parents. Les opérateur
        génétique son appliqué ici.

        Entrée : une liste de parents (cf.  make_parent_pop)
        Sortie : une liste de solution enfants de taille self._lambda

        """
        offspring = []
        done = False
        while not done :

            # On sélectionne les géniteurs
            y,z = self.tournament_selection(parents,2,5)

            y1 = y
            z1 = z
            # On a une certaine probabilité de croisement
            if (pc > random.uniform(0, 1)):
                feasable=False
                # On cherche a obtenir des solutions enfants qui sont valides
                while (not feasable):
                    childs=self.bit_point_crossing(y,z)
                    y1=childs[0]
                    z1=childs[1]
                    feasable=self._problem.feasable(y1) and self._problem.feasable(z1)
                self._problem.eval(y1)
                self._problem.eval(z1)

            # On a une certaine probabilité de mutation
            # On cherche a obtenir des mutations qui sont valides
            if (pm > random.uniform(0, 1)):
                feasable=False
                while (not feasable):
                    y1=self.bit_uniform_mutation(y1)
                    feasable=self._problem.feasable(y1)
                feasable=False
                while ( not feasable):
                    z1=self.bit_uniform_mutation(z1)
```

```
                    feasable=self._problem.feasable(z1)
                self._problem.eval(y1)
                self._problem.eval(z1)

            # on a rajoute a la liste
            # les parents
            offspring.append(y)
            offspring.append(z)
            # et les enfants
            offspring.append(y1)
            offspring.append(z1)

            # on arrête si la population est remplie
            done = len(offspring) == self._lambda

        return offspring

    def make_new_pop(self, offspring):
        """
        Constituer la nouvelle population selon une stratégie de selection
        donnée depuis les enfants et self._pop.

        Entrée : une liste de solutions enfants
        Sortie : une liste de solutions de taille self._mu

        """

        # On tri les enfants
        self.sort_pop(offspring)

        # on garde les mu meilleurs
        survivors = []
        for i in xrange( self._mu ):
            survivors.append( offspring[i] )
        return survivors
```