TP 5 – Métaheuristique

Les nombres de Schur

Bellanger Clément

Questions

Dans chaque instance des nombres de Schur, les valeurs k et n (3 et 13 respectivement dans l'exemple)

Espace de recherche

Une solution est une liste de k-vecteurs de n-booléens, ou chaque booléen reflète si le nombre i/n du vecteur appartient à la partition (la liste) j/k

Illustration pour S(3)

Si on considère 0 comme False et 1 comme True.

[1,0,0,1,0,0,0,0,0,1,0,0,1]

[0,1,1,0,0,0,0,0,0,0,1,1,0]

[0,0,0,0,1,1,1,1,1,0,0,0,0]

Fonction objectif

Pour la fonction objectif, la valeur à minimiser est le nombre de sommes incorrectes :

Par exemple pour :

{1,2,3,4}

{5}

On aura la valeur objectif égale à 3 car 2=1+1 3=2+1 et 4=2+2 ou 4=3+1

La valeur objectif 0 correspond donc à une solution valide

Opérateur de voisinage

On a utilisé comme opérateur de voisinage l'inversion de deux nombres dans deux listes différentes.

(Il semblerait après coup que cela ne soit pas la meilleure solution car on obtient uniquement des listes de même taille)

Implémentation

Pour l'implémentation du problème on a choisi pour une instance de type small de fixer k a 3 et n a 13 afin de tester une solution faisable. Pour une instance medium k:5 et n:160 et large k:6 n:400.

Il est également possible d'utiliser un k et un n défini en les spécifiant en paramètre de generate_schur_instance().

On a créée une classe spécifique dans solution : VectorBinarySolution

Pour la fonction objectif on traduit d'abord les listes de booléen en listes d'entiers (telles que l'on peut les voir dans l'affichage d'une solution).

A l'aide des listes d'entiers on crée une seconde liste globale de booléens qui contient les sommes interdites (pour chaque liste).

On compare les listes de sommes interdites avec les listes de booléens initiaux et on augmente l'objectif de 1 pour chaque valeur vraie en commun.

Résultats.

Les tests on été effectué sur trois algorithmes.

Comme les tests prennent beaucoup de temps sur ce problème, ils ont été effectués en taille small

RandomLS

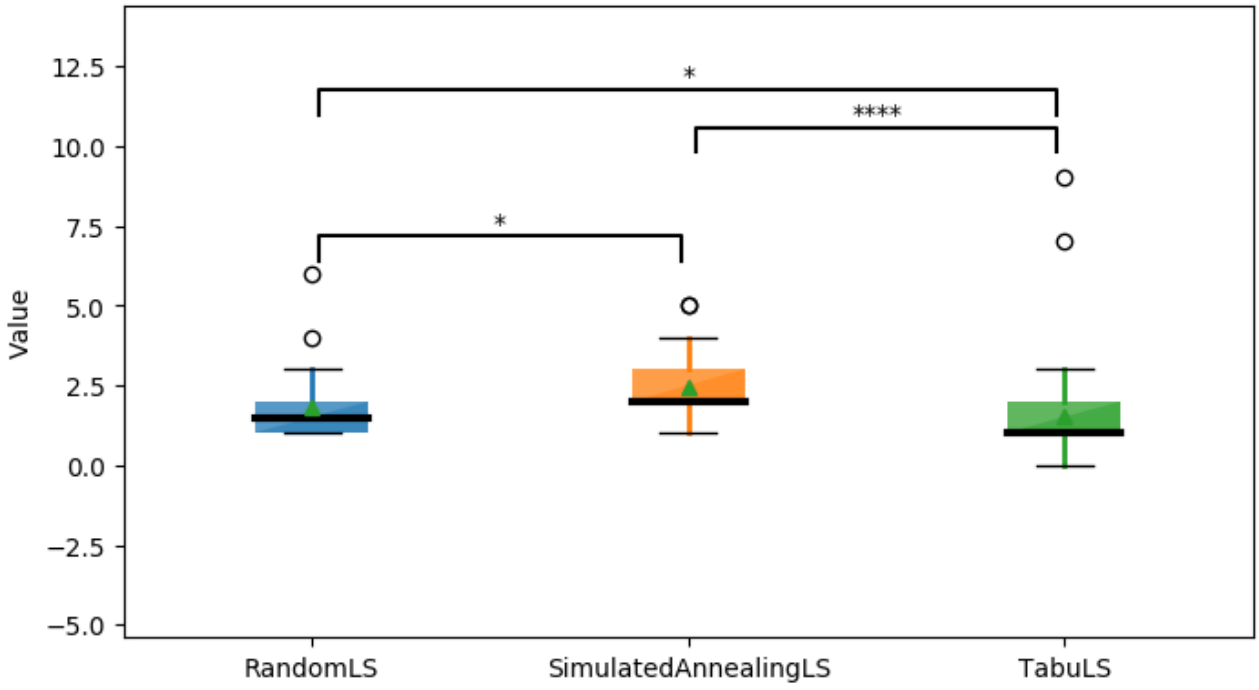
SimulatedAnnealingLS

Température initiale : 300

gamma : 0.99

TabuLS

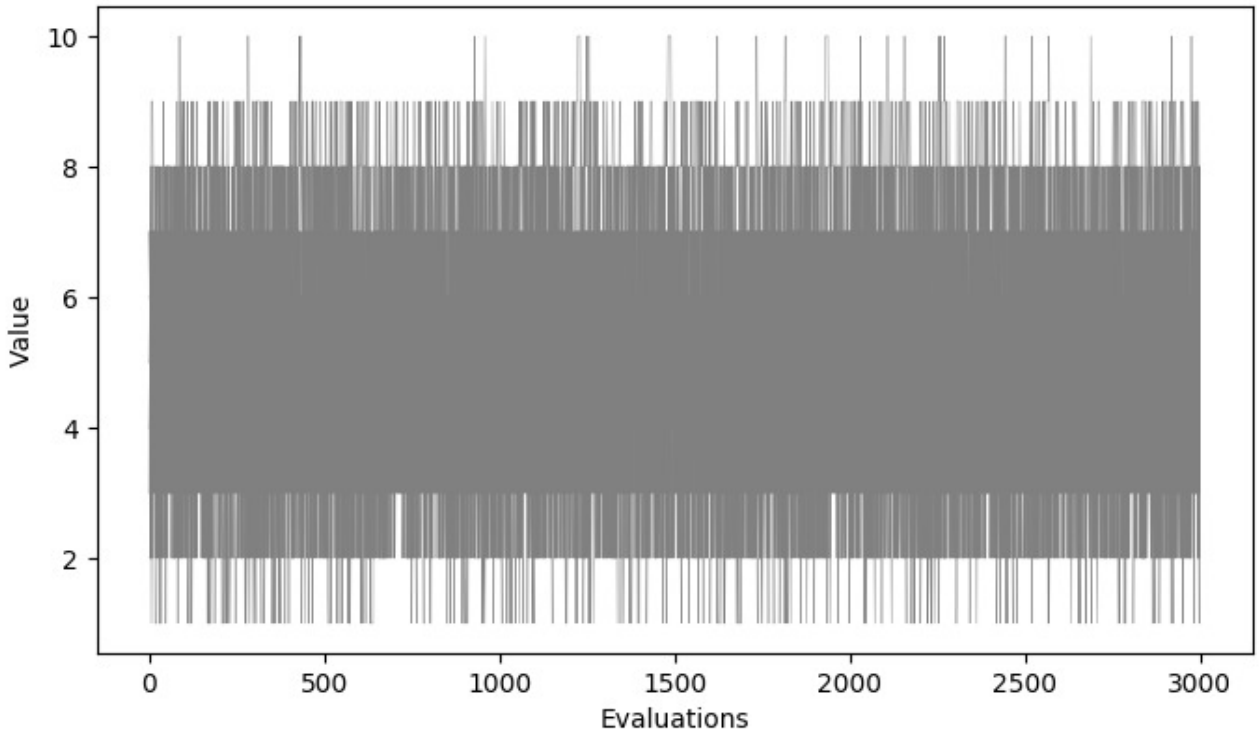
taille de la liste : 50


12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
Value
*
****
*
RandomLS
SimulatedAnnealingLS
TabuLS

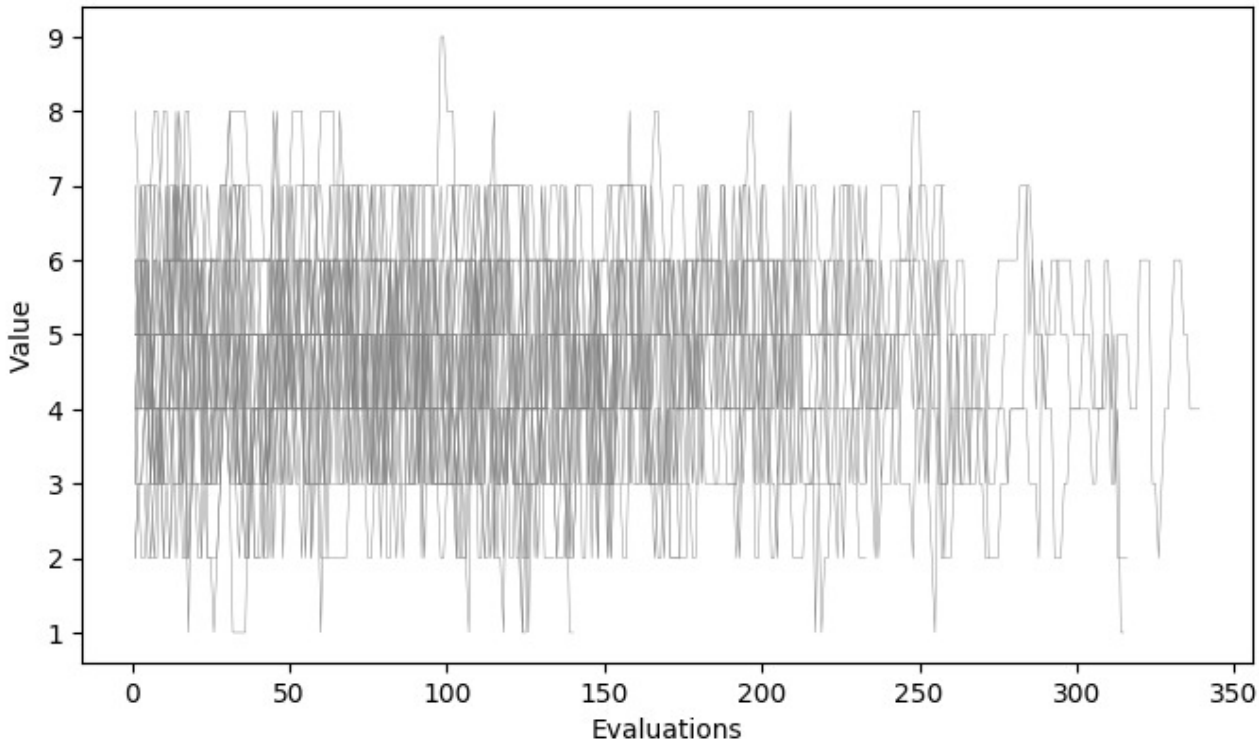
Problème – Schur

Trace des algorithmes (taille large)

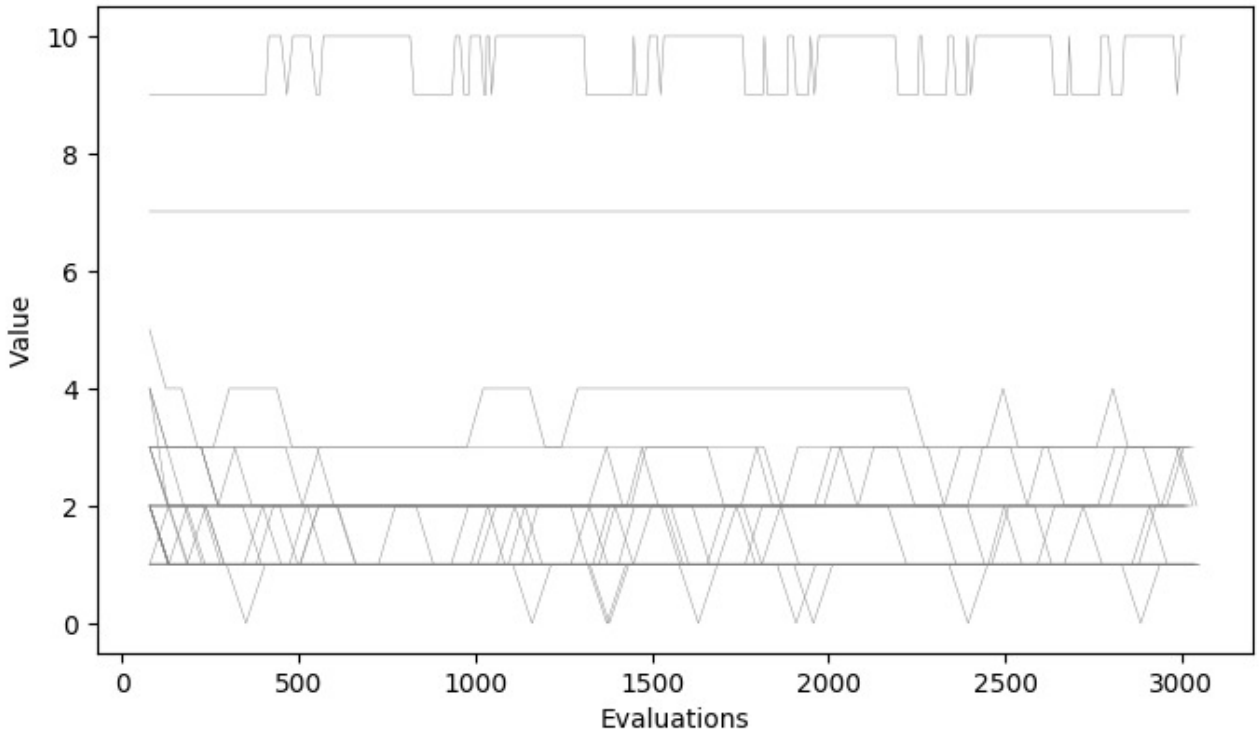
RandomLS



SimulatedAnnealingLS



TabuLS



Commentaire

On constate que l’algorithme de recherche taboue parvient à trouver la solution contrairement aux autres.

Mais ce résultat est à relativiser, en effet pour avoir une chance de trouver la bonne solution il faut déjà tomber sur le bon nombre de numéros dans chaque liste (voir fonction de voisinage).

Mais on peut affirmer que dans ce cas (peu de solutions à tester) les recherches aléatoires et taboues donnent généralement de bon résultats.

Code commenté

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

"""
La sous-classe de Problem pour les nombres de Schur

Une solution est une liste de k-vecteurs de n-booléens, ou chaque booléen
reflète si le nombre i/n du vecteur appartient à la partition (la liste) j/k

"""

class Schur(Problem) :

    def __init__(self, k, n, max_eval=1000):
        Problem.__init__(self, max_eval)
        self._name = "Schur"
        self._minimize = True
        self._k=k
        self._n=n

    def feasable(self, sol) :
        """
        On considère qu'une solution est faisable lorsque chaque nombre se trouve dans une et une
seule liste.
        Les sommes seront utilisé pour la fonction de fitness.
        """
        if not isinstance(sol, VectorBinarySolution):
            raise TypeError("x must be a instance of Solution")

        for i in xrange(len(sol._sol[0])):
            # Un nombre ne doit etre présent que dans une seule liste
            # On retourne faux si il est présent dans plusieurs listes ou dans aucune
            isInColumn=False
            for li in xrange(len(sol._sol)):
                if sol._sol[li][i] == True:
                    # Faux, il est présent dans plusieurs listes
                    if isInColumn == True:
                        return False
                    else:
                        # Le nombre est présent au moins une fois dans une liste
                        isInColumn=True
            # Faux, après le parcours de la colonne, le nombre n'est dans aucune liste
            if isInColumn == False:
                return False

        # Après le parcours de la ligne, on a vérifié toutes les colonnes, donc la solution est valide
        return True

    def setForbidden(self, forbiddenTable, forbiddenNumber):
        if forbiddenNumber <= len(forbiddenTable):
            forbiddenTable[forbiddenNumber-1]=True

    def eval(self, sol):
        if not isinstance(sol, VectorBinarySolution):
            raise TypeError("x must be a instance of Solution")

        self.nb_evaluations += 1
        # On traduit la liste de booléens en liste d'entier (en fonction des valeurs à True)
        liste_globale = []
        for i in xrange(len(sol._sol)):
            liste_locale = []
            for j in xrange(len(sol._sol[i])):
                if (sol._sol[i][j]):
                    liste_locale.append(j+1)
            liste_globale.append(liste_locale)

        #On crée un tableau à double entrée pour noter les nombres interdits (de meme taille avec
booléen)
        forbidden_global = np.zeros((self._k,self._n), dtype=np.bool_)
```

```
        for i in xrange(len(liste_globale)):
            for j in xrange(len(liste_globale[i])):
                for k in xrange(j+1, len(liste_globale[i])):
                    #print(forbidden_global[i])
                    self.setForbidden(forbidden_global[i],liste_globale[i][j]*2)
                    self.setForbidden(forbidden_global[i],liste_globale[i][k]*2)
                    self.setForbidden(forbidden_global[i],liste_globale[i][j]+liste_globale[i][k])
        #print(forbidden_global)

        fitness=0
        # On refait un parcours de toute la solution et on ajoute 1 à chaque somme interdite
        for i in xrange(len(sol._sol)):
            for j in xrange(len(sol._sol[i])):
                if (sol._sol[i][j] == True and forbidden_global[i][j] == True):
                    fitness+=1
        sol._value=fitness
        return fitness

    def print_solution(self, sol):
        if not isinstance(sol, VectorBinarySolution):
            raise TypeError("x must be a instance of Solution")
        return "val:{} sol:\n{}".format(sol._value,str(sol))


    def generate_initial_solution(self, sol_type='empty'):

        if sol_type not in [ 'empty', 'random' ] :
            raise ValueError("Unknown inital solution type")

        initial_solution = VectorBinarySolution(dim=self._k)
        initial_solution._sol = np.zeros((self._k,self._n), dtype=np.bool_)
        for i in xrange(self._n):
            randomInt=np.random.randint(self._k)
            initial_solution._sol[randomInt][i]=True
        self.print_solution(initial_solution)
        return initial_solution



"""

Factory pour generer des problèmes de schur

"""

def generate_schur_instance(prob_type, max_eval, k=-1, n=-1):
    """
    Pour générer une instance du problème des nombres de Schur

    prend :
       prob_type : type 'small', 'medium', 'large', 'random'
       max_eval : le nombre d'evaluations maximum alloué
       k : le nombre de listes (optionnel uniquement si type est random)
       n : le nombre à tester (optionnel uniquement si type est random)

    retourne : une instance de la classe Schur

    """

    if prob_type not in ['small', 'medium', 'large', 'random'] :
        raise ValueError("Unknown prob_type instance")

    volume = 0
    items = []

    if prob_type is 'small' :
        #Pour tester  fitness:0 (solution acceptable), k:3, n:13
        #[1,0,0,1,0,0,0,0,0,1,0,0,1]
        #[0,1,1,0,0,0,0,0,0,0,1,1,0]
        #[0,0,0,0,1,1,1,1,1,0,0,0,0]
        k=3
```

```
        n=13


    elif prob_type is 'medium' :
        #Pour tester  fitness:0 (solution acceptable), k:5, n:160
        k=5
        n=160

    elif prob_type is 'large' :
        k=6
        n=400

    elif prob_type is 'random' :
        if k == -1:
            k=np.random.randint(5, high=101)
        if n == -1:
            k=np.random.randint(160, high=1000)


    problem = Schur(k, n, max_eval=max_eval)

    return problem
```

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

"""
Classe abstraite a concrétiser, voire plus bas.

"""

class Solution(object):

    def __init__(self, dim=None, x=None):
        """ constructeur presque vide  voire les classe concrètes plus bas """
        if dim is None and x is None :
            raise ValueError("Il faut spécifier la dimension ou une solution")

        if dim is None :
            self._dim = len(x)
        else :
            self._dim = dim
        if x is not None :
            self._sol = np.copy(x)

        self._value = None

    @property
    def solution(self):
        return self._sol

    @property
    def value(self):
        return self._value

    def random(self):
        """
        crée une solution aléatoire
        retourne un instance de Solution
        """
        raise NotImplementedError

    def neighbors(self):
        """
        permet de construire l'ensemble des solutions voisines de la solution
        courante
        retourne ensemble solutions voisines de this
        """
        raise NotImplementedError

    def clone(self):
        """ clone la solution dans une nouvelle instance """
        raise NotImplementedError

    def __str__(self):
        """ une méthode to string """
        raise NotImplementedError

    def __eq__(self, other):
        """
        une méthode pour vérifier l'égalité de deux solutions
        Il faut la même dimention et que les tableaus soient identiques

        """
        if  self._dim != other._dim :
            return False
        if self._sol.shape != other._sol.shape :
            return False
        return (self._sol == other._sol).all()

"""
Classe abstraite pour representer une solution comme un tableau binnaire
```

```
"""

class BinarySolution(Solution):

    def __init__(self, dim=None, x=None):
        Solution.__init__(self, dim, x)
        if x is None :
            self._sol = np.zeros(dim, dtype=np.bool_)

    def random(self):
        """ Retourne une solution aléatoire """
        rnd = np.random.random(self._dim) < 0.5
        return BinarySolution(x=rnd)

    def neighbors(self):
        """ Retourne toutes les solutions voisines i.e. differentes de 1 """
        N = []
        for i in xrange(len(self._sol)):
            n = np.copy( self._sol )
            n[i] = not n[i]
            N.append(BinarySolution(x=n))

        # mélanger pour rendre le parcours non déterministe
        shuffle(N)
        return N

    def clone(self):
        """ Pour cloner la solution """
        clone_sol =  BinarySolution( x=self._sol )
        clone_sol._value = self.value
        return clone_sol

    def __str__(self):
        """ une méthode to string pour afficher la solution """
        return "".join([ '1' if i else '0' for i in  self._sol ])



"""
Classe abstraite pour representer une solution comme un tableau de permutation

"""

class PermutationSolution(Solution):

    def __init__(self, dim=None, x=None):
        Solution.__init__(self, dim, x)
        if x is None :
            self._sol = np.zeros(dim, dtype=np.int)

    def random(self):
        """ Retourne une solution aléatoire """
        rnd = np.arange(self._dim, dtype=np.int)
        np.random.shuffle(rnd)
        return PermutationSolution(x=rnd)

    def neighbors(self):
        """
        Retourne toutes les solutions voisines i.e. differentes de 1
        échanger 2 elements du tableau

        """
        N = []
        for i in xrange(len(self._sol)):
            for j in xrange(i+1,len(self._sol)):
                if i != j :
                    n = np.copy( self._sol )
                    tmp = n[i]
                    n[i] = n[j]
                    n[j] = tmp
                    N.append( PermutationSolution (x=n) )
```

```
        # mélanger pour rendre le parcour non déterministe
        shuffle(N)
        return N

    def clone(self):
        """ Pour cloner la solution """
        clone_sol = PermutationSolution( x=self._sol )
        clone_sol._value = self.value
        return clone_sol

    def __str__(self):
        """ une méthode to string pour afficher la solution """
        return ",".join([ str(i) for i in  self._sol ])




"""
Classe abstraite pour representer une solution comme un vecteur de réels

"""

class RealSolution(Solution):

    def __init__(self, dim=None, x=None):
        Solution.__init__(self, dim, x)
        if x is None :
            self._sol = np.zeros(dim, dtype=np.float_)

    def random(self):
        """ Retourne une solution aléatoire dans [-5, 5]^dim """
        rnd = np.random.random(self._dim)
        rnd *= 10
        rnd -= 5
        return RealSolution(x=rnd)

    def neighbors(self):
        raise NotImplementedError("Solution réelles n'ont pas de voisinage")

    def clone(self):
        """ Pour cloner la solution """
        clone_sol =  RealSolution( x=self._sol )
        clone_sol._value = self.value
        return clone_sol

    def __str__(self):
        """ une méthode to string pour afficher la solution """
        return ",".join([ str(i) for i in  self._sol ])

"""
Classe abstraite pour representer une solution comme plusieurs vecteurs binaire
Utilisé par le problème des nombres de Schur

"""

class VectorBinarySolution(Solution):

    def __init__(self, dim=None, x=None):
        Solution.__init__(self, dim, x)
        #self._sol=np.zeros((dim,len(x._sol[0])), dtype=np.bool_)
        if x is None :
            self._sol = np.zeros((dim,10000), dtype=np.bool_)
            for i in xrange(10000):
                randomInt=np.random.randint(dim)
                self._sol[randomInt][i]=True

    def random(self):
        """ Retourne une solution aléatoire """
        # On connait déjà les solutions pour n <= 4 donc on commence a 5
        randomInt=np.random.randint(5, high=101)
        #rnd = np.random.random(self._dim) < 0.5
        return VectorBinarySolution(dim=randomInt)
```

```
        #0,1,0,1    i: 0 < 4 j: i+1 < 4
        #                   i:0 j: 1 < 4 (1,2,3)
        #                   i:1 j: 2 < 4 (2,3)
        #                   i:2 j: 3 < 4 (3)
        #                   i:3 j: 4 < 4 X
        #1,0,0,0
        #0,0,1,0

        #i 1ère permutation j 2ème permutation
        #l1 1ère liste l2 2ème liste
        #for li < dim li++
        #    if li[i] == True:
        #        l1=li
        #    elif li[j] == True:
        #        l2=li

        #swap(permut1,permut2,l1,l2)
        #temp=l1[permut1];
        #l1[permut1]=l2[permut2]
        #l2[permut2]=temp



    def neighbors(self):
        """ Retourne toutes les solutions voisines i.e. differentes de 1 """
        N = []
        for i in xrange(len(self._sol[0])):
            for j in xrange(i+1, len(self._sol[0])):
                # On essaie de trouver l1 et l2, les listes à permuter
                l1=-1
                l2=-1
                for li in xrange(len(self._sol)):
                    if self._sol[li][i] == True:
                        l1=li
                    if self._sol[li][j] == True:
                        l2=li
                n = np.copy( self._sol )
                new_solution=VectorBinarySolution(x=n)
                new_solution.swap(i,j,l1,l2)
                N.append(new_solution)

        # mélanger pour rendre le parcours non déterministe
        shuffle(N)
        return N


    def swap(self,permut1,permut2,l1,l2):
        """ Permute deux listes aux indices l1 et l2 à la position permut1 et permut2
respectivement."""
        self._sol[l1][permut1]=False
        self._sol[l2][permut2]=False
        self._sol[l1][permut2]=True
        self._sol[l2][permut1]=True

    def clone(self):
        """ Pour cloner la solution """
        clone_sol =  VectorBinarySolution( x=self._sol )
        clone_sol._value = self.value
        return clone_sol

    def __str__(self):
        """ une méthode to string pour afficher la solution """
        string=""
        for i in xrange(len(self._sol)):
            string+="["
            vide=True
            for j in xrange(len(self._sol[i])):
                if (self._sol[i][j]):
                    string+=str(j+1)
                    string+=","
                    vide=False
```

```
            # On enlève la virgule finale si besoin
            if vide == False:
                string=string[:-1]
            string+="]\n"
        return string
```