Étude des solutions d'un casse-tête en 3D

Projet ZZ1

Jouseau Roxane, Grenier Charline
Responsable de projet : Dalmas Benjamin

Année scolaire 2017/2018

ISIMA
Institut Supérieur d'Informatique de Modélisation et de leurs Applications

Table des matières

1 Présentation du problème **3**

2 Modélisation **4**

3 Méthode de résolution **4**

4 Mise en place de l'algorithme génétique **5**
4.1 Pondération . . . 6
4.2 Croisement . . . 6
4.3 Mutation . . . 6

5 Résultats obtenus **7**

6 Analyse des résultats **8**
6.1 Résultats obtenus . . . 8
6.2 Améliorations possibles . . . 8
6.2.1 Choix du croisement . . . 8
6.2.2 Optimisation de l'algorithme en général . . . 8

7 Difficultés rencontrées **8**
7.1 Dans la modélisation . . . 8
7.2 Dans l'optimisation du code . . . 9

8 Rétrospective sur le travail fourni **9**

9 Codes python **10**
9.1 Fichier des cubes . . . 10
9.2 Valeurs initiales . . . 10
9.3 Appel des fonctions . . . 10
9.4 Algorithme génétique . . . 11
9.5 Définition des cubes . . . 12
9.6 Fonctions de gestion des cubes . . . 13
9.7 Fonctions de gestion des figures . . . 17
9.8 Pondération des figures . . . 18
9.9 Mutation d'une figure . . . 18
9.10 Croisement de deux figures . . . 19
9.11 Tirage pondéré . . . 20

1 Présentation du problème

Nous avons choisi de travailler sur un casse-tête en 3 dimensions constitué de 27 petits cubes s'emboîtant entre eux ce qui permet de construire différentes formes, la plus complexe étant un grand cube de 3x3x3. Chaque petit cube présente une prise mâle ou femelle par face (il existe 4 formes de prises différentes, 8 en différenciant les prises mâles des prises femelles). Notre première problématique était de trouver toutes les combinaisons de cubes possibles qui pouvaient former le cube 3x3x3 pour lequel nous n'avions qu'une solution inhérente à la création du casse-tête. Cependant nous voulions implémenter une méthode métaheuristique qui étant non déterministe par définition ne pouvait pas donner une liste exhaustive des solutions avec certitude. Nous nous sommes donc penchées sur une autre facette du problème à savoir : trouver au moins une construction pour n'importe quelle figure entrée en argument et étant constructible (aucune étude préalable sur la question nous indique que toute figure constituée de 27 cubes ou moins est constructible avec notre set de cubes).

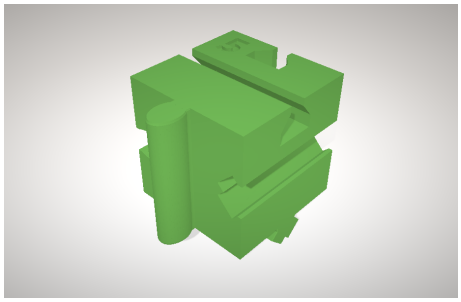

Figure 1 – cube 5

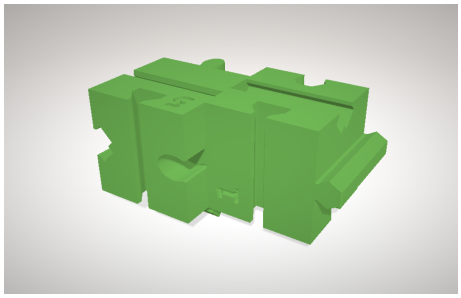

Figure 2 – cubes 5 et 1

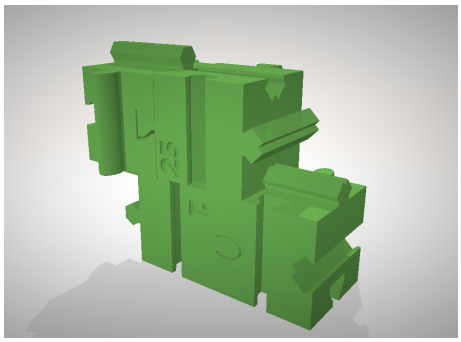

Figure 3 – cubes 5, 1, 25 et 16

2 Modélisation

Nous avons modélisé une figure en 3 dimensions par une matrice de taille 3x3x3 dans laquelle nous plaçons des objets cubes. Un cube étant défini à une position de référence chacune de ses faces comportant une prise mâle ou femelle (il existe 4 formes de prises différentes, 8 en différenciant les prises mâles des prises femelles). Chaque forme de prise est représentée par un entier entre 1 et 4 pour les prises mâles et -1 et -4 pour les femelles correspondantes. Ainsi deux faces sont compatibles si la somme de leurs prises est nulle. Afin de ne pas travailler avec des matrices 3x3x3 contenant des cubes ce qui serait très coûteux en mémoire nous avons choisi de travailler avec un dictionnaire de cubes. Chacun des cubes étant de toutes façons physiquement numéroté il ne restait qu'à les rentrer dans un fichier avec les numéros de leurs prises dans l'ordre des faces pré-établi. Une position de référence ayant aussi été choisie de façon arbitraire pour l'ensemble des cubes (numéro face caché).

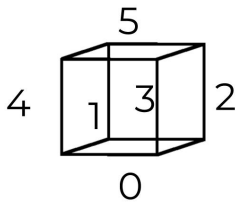

Figure 4 – numérotation des faces d'un cube

3 Méthode de résolution

Nous avons décidé d'utiliser un algorithme génétique pour résoudre ce problème car toute figure finale est obligatoirement formée de sous figures qui sont donc solutions du problème avec pour entrée une figure de taille inférieure à la figure souhaitée.

Les algorithmes génétiques ont été introduits pour la première fois par John Holland en 1960 et ont été popularisés par David Goldberg (dans le livre "Genetic Algorithms in Search, Optimization, and Machine Learning" en 1989). Le but des algorithmes génétiques est de trouver une solution approchée à un problème d'optimisation. Ils permettent donc d'obtenir une solution relativement satisfaisante à un problème lorsque explorer toutes les solutions est impossible (parce que la complexité est trop grande ou que la méthode d'exploration n'est pas exacte). Ces types d'algorithmes s'appuient donc sur une analogie de la théorie de l'évolution et de la sélection naturelle qui pourrait être résumée grossièrement par : la combinaison de solutions dans le voisinage de la solution optimale tends vers la solution optimale, quand la distance entre la solution et la figure observée diminue. On reprendra donc 3 étapes importantes en analogie avec la biologie :

— Le croisement entre deux individus,
— La mutation possible du nouvel individu,
— La viabilité du nouvel individu

En effet, tout comme certaines modifications dans le génotype non viables aboutissent à la mort de l'individu nous éliminerons les combinaisons de cubes ne respectant plus les contraintes du problème que nous nous sommes fixé. Il est aussi possible d'ajouter un critère dit d'élitisme (ce que nous ferons). Ce critère assure le fait qu'un nouvel élément ne remplace ses parents dans la génération suivante si et seulement si il est considéré meilleur que ses parents par rapport à la (ou les) caractéristique(s) que l'on cherche à améliorer (c'est-à-dire la distance entre l'objet et l'optimum local a diminuée).

Ainsi, dans le cadre de notre problème spécifique combiner différentes petites figures pour former une nouvelle figure plus grande semble être une bonne façon de résoudre le problème. De plus un algorithme déterministe explorant toutes les combinaisons différentes serait bien trop long en temps d'exécution sur des figures de grande taille.

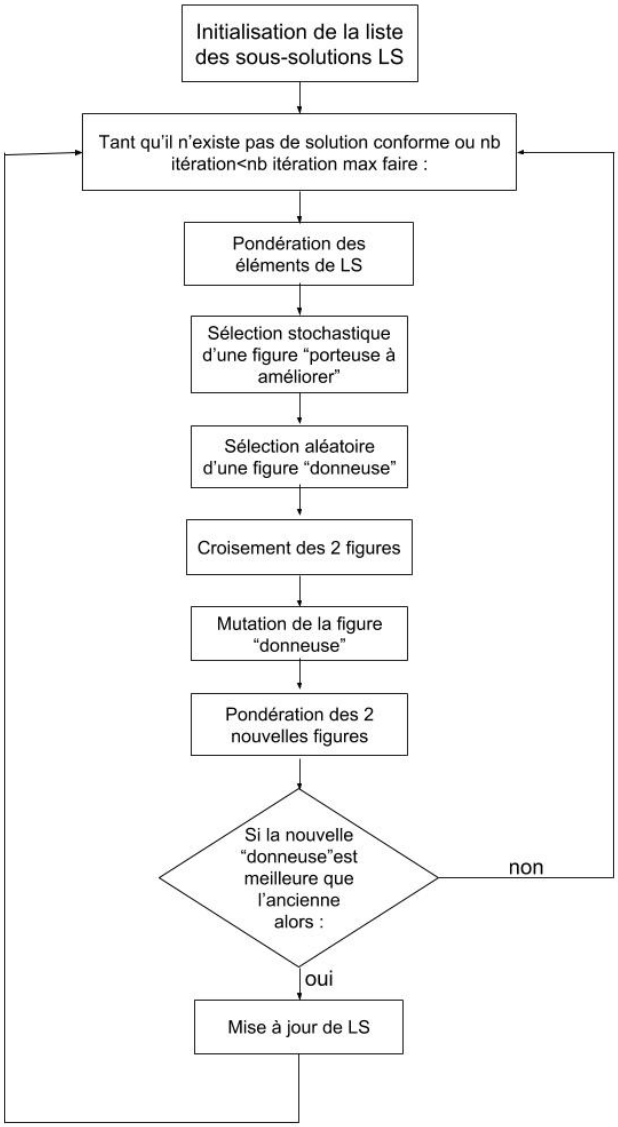

Figure 5 – Schéma de principe de l'algorithme génétique

4 Mise en place de l'algorithme génétique

Pour mettre en place l'algorithme génétique il est nécessaire de définir 3 éléments principaux :

- La pondération (Comment définit-on la proximité d'une sous-solution à une solution locale existante ou non ?)
- Le croisement (Comment croise-t-on 2 sous-solutions ?)
- La mutation (Comment définit-on la mutation d'une sous-solution ?)

4.1 Pondération

Nous cherchons ici à construire une figure en 3 dimensions comportant N cubes ($N \leq 27$) un optimum local est donc une figure comportant N cubes, viable (dont les faces sont compatibles entre elles dans la disposition choisie) et respectant les critères de hauteur, largeur et longueur donnés en paramètres. Si on élimine donc les figures possédant des cubes hors limites définies et des problèmes de compatibilité de faces en les considérant comme non viables on peut convenir de manière naturelle que plus le nombre de cubes d'une figure se rapproche de N, plus la figure se rapproche de l'optimum local. Nous prendrons donc le nombre de cubes présents dans une figure comme facteur de pondération. Chaque figure sera initialisée avec un unique cube différent (la pondération ne sera donc pas nulle à l'initialisation). Il est cependant important de faire remarquer que l'optimum local auquel on compare la sous-solution n'existe pas toujours. En effet, il n'est en aucun cas assuré qu'il soit possible de construire la figure voulue avec notre set de cubes ou l'initialisation d'une figure spécifique.

4.2 Croisement

Passons maintenant à la définition d'un croisement entre deux figures de la liste des sous-solutions. L'objectif est d'obtenir une figure comportant des caractéristiques de ses deux parents tout en se rapprochant de l'optimum local. C'est à dire combiner des cubes des deux solutions et former une nouvelle solution comportant plus de cubes qu'au moins une des deux solutions (un fils meilleur qu'au moins un de ses deux parents). Cependant étant donné le nombre très élevé de contraintes auxquelles nous sommes soumis (correspondances de toutes les faces de chaque cube vis à vis de tous ses voisins et respect des limites de la figure) il serait très coûteux de recréer une figure à chaque itération de l'algorithme nous nous contenterons donc de choisir une figure "porteuse" à laquelle nous ajouterons un cube prélevé dans une figure "donneuse". La figure "porteuse" sera sélectionnée de façon pseudo-aléatoire, un tirage aléatoire déterminera quelle figure sera choisie mais les figures avec le plus grand nombre de cubes auront le plus de probabilité d'être choisies. Quand à la figure "donneuse" elle sera choisie de façon totalement aléatoire. On peut encore une fois noter que le croisement entre deux figures n'est pas toujours possible, en effet il peut arriver qu'aucun cube de la figure donneuse ne soit compatible avec un emplacement de la figure porteuse.

4.3 Mutation

La mutation elle non plus n'est pas toujours possible. On définit la mutation d'une figure par le déplacement d'un de ses cubes vers une position non occupée dans la figure.

On peut noter que comme la liste des figures sous-solutions est initialisée avec un cube différent dans chaque figure, les figures de la liste restent disjointes deux à deux sur leur composition en cubes ce qui empêche les conflits sur l'utilisation d'un même cube dans plusieurs positions de la figure et réduit donc les tests à faire lors des croisements.

5 Résultats obtenus

```
nombre d'iteration : 87
solution :
[[[5], [25]], [[3], [15]]]
cubes avec leur faces
5 : [4, -4, -2, -1, 1, -2]
25 : [2, 4, -4, 2, -4, -3]
3 : [-4, 2, -2, -4, 3, 4]
15 : [-4, -4, 3, -1, 4, -3]
```

Figure 6 – Résultat pour une figure 2x2x1
L'algorithme a trouvé une solution et retourne les numéros des cubes à placer dans la matrice et la liste de leurs faces pour avoir leurs orientations dans la figure.

```
nombre d'iteration : 13861
solution :
[[[20], [10]], [[6], [19]], [[4], [25]]]
cubes avec leur faces
20 : [1, -3, 3, -2, 4, -4]
10 : [-2, 1, 3, 4, 2, 2]
6 : [1, -2, -4, -1, 4, 4]
19 : [-4, -3, -1, 4, -1, 4]
4 : [-1, 1, 4, 2, -4, -3]
25 : [2, 4, -4, 2, -4, -3]
```

Figure 7 – Résultat pour une figure 3x2x1
L'algorithme a trouvé une solution et retourne les numéros des cubes à placer dans la matrice et la liste de leurs faces pour avoir leurs orientations dans la figure.

```
nombre d'iteration : 1000000
pas de solution trouvee
```

Figure 8 – Résultat pour une figure 2x2x2
L'algorithme n'a pas trouvé de solution. Le nombre nombre d'itérations maximum a été atteint.

6 Analyse des résultats

6.1 Résultats obtenus

Pour des figures simples en 2 dimensions, l'algorithme n'a aucun problème à trouver une solution assez rapidement. Cependant dès lors que l'on demande des figures plus complexes il ne parvient pas à aboutir à un optimum.

Nous avons estimé que 100 000 itérations du "while" prenaient en moyenne 10 secondes. Cependant, après avoir fait tourner l'algorithme pendant 24 heures consécutives (soit environ un peu plus de 14 millions d'itérations) sur une figure de taille 2x2x2, il ne trouvait toujours pas de solutions.

6.2 Améliorations possibles

6.2.1 Choix du croisement

Nous avons ici considéré qu'un croisement était le déplacement d'un cube d'une figure A vers une figure B. Cependant une deuxième méthode plus proche de la réalité biologique pourrait être de prendre un certains nombre de cubes provenant des figures A et B pour en former une nouvelle mais cette méthode est bien plus complexe à implémenter et présente une complexité temporelle bien plus élevée. On aurait aussi pu considérer plus simplement de déplacer un bloc de cubes de la figure A vers la figure B (comme une ligne ou une colonne) mais par manque de temps nous nous sommes limités au déplacement d'un unique cube.

De plus différentes sources évoquent le fait que souvent un croisement avec un nombre de parents supérieur à deux était plus favorable à une convergence rapide des sous solutions vers un optimum local. Le nombre de parents lors d'un croisement pourrait donc être un paramètre intéressant à faire varier.

6.2.2 Optimisation de l'algorithme en général

Nous suspectons aussi que nos algorithmes pourraient être améliorés pour gagner en complexité temporelle et/ou physique. En effet ils ont tous été écrits une première fois pour effectuer les tests mais nous n'avons pas eu le temps de les revoir par la suite.

On pourrait aussi proposer de faire varier l'initialisation de la liste des sous-solutions. En effet, l'algorithme trouve facilement des solutions pour des figures en 2 dimensions mais peine à trouver des solutions pour des figures en 3 dimensions. Il pourrait donc être pertinent d'initialiser la liste des sous-solutions avec des solutions trouvées pour la base de la figure en 3 dimensions. Lors d'une recherche pour une figure de taille 2x2x2 on initialiserait donc la liste des sous-solutions avec des solutions précédemment trouvées pour des figures de taille 2x2x1.

7 Difficultés rencontrées

7.1 Dans la modélisation

Ce projet ayant pour sujet un casse-tête en 3 dimensions avec une méthode de résolution sur laquelle nous n'avions jamais travaillé auparavant les difficultés que nous avons rencontrées ont été nombreuses.

Tout d'abord il nous a fallu modéliser le problème de A à Z ce qui ne fut pas aisé, nous avons défini plusieurs fois l'objet cube avec différents critères et approches avant d'en choisir une qui nous semblait la plus adaptée à notre utilisation.

Ensuite il a fallu déterminer comment nous allions faire correspondre les fonctions de croisements et de mutations de l'algorithme génétique sur nos figures. Les figures étant soumises à un nombre de

contraintes assez élevé nous devions tenir compte de la complexité de ces fonctions qui augmentait très rapidement.

De plus définir une pondération des figures pour les comparer n'a pas non plus été évident, il fallait être sûr que le critère augmentait quand la figure se rapprochait d'un potentiel optimum local et qu'il n'était pas trop coûteux en complexité à calculer car il est souvent appelé dans l'algorithme.

7.2 Dans l'optimisation du code

Une fois l'algorithme implémenté nous nous sommes rendu compte qu'il peinait voire n'arrivait pas à trouver de solution optimale pour des figures de plus d'un étage (donc en 3 dimensions). Faute de temps nous n'avons pas pu résoudre ce problème mais nous pensons que les modifications à apporter sont relatives à l'optimisation de nos algorithmes déjà implémentés.

8 Rétrospective sur le travail fourni

Tant l'objet étudié que la méthode de résolution étaient nouveaux pour nous, ce projet nous a donc permis de travailler et d'acquérir diverses compétences qui ne sont pas forcément abordées en première année à l'ISIMA.

Ainsi nous avons pu travailler sur la façon de passer d'un problème concret et modéliser ses contraintes et ses acteurs de façon adaptée pour pouvoir ensuite travailler avec.

Nous avons aussi choisi de réaliser le projet en orienté objet avec python ce qui nous a donc permis d'approfondir nos compétences relatives à ce langage et à l'orienté objet spécifiquement.

Il nous a aussi fallu comprendre les tenants et les aboutissants des algorithmes génétiques et plus généralement le but et les concepts des métaheuristiques. Beaucoup de recherches et de lecture de documentations annexes ont donc été nécessaires à la réalisation de ce projet ce qui nous a aussi permis d'enrichir notre culture générale scientifique dans un domaine précis que nous ne connaissions pas ou peu.

9 Codes python

9.1 Fichier des cubes

```
20;-2,3,-4,4,1,-3
7;-2,-3,-2,1,-3,2
24;-1,4,-2,-3,3,2
2;-1,2,4,1,-3,2
5;-2,1,-1,-2,-4,4
9;-4,2,-3,4,-2,3
11;-4,2,-4,2,4,-2
17;-2,-4,-1,4,-4,1
19;-4,-1,-3,-1,4,4
21;-2,3,-1,-4,-4,4
23;-1,2,-1,-3,2,-3
1;-2,3,-1,1,-4,-3
3;-4,-4,3,2,-2,4
4;-4,-1,2,-3,1,4
6;-1,-4,4,4,1,-2
8;-2,-2,3,-1,3,1
10;-2,2,1,3,4,2
12;-3,-4,-4,-2,4,3
13;-1,-3,-4,-2,3,1
14;-4,-4,-3,-2,4,1
15;-4,3,-4,4,-3,-1
16;-3,1,4,3,1,-2
18;-1,4,4,-3,3,-3
22;-4,-2,2,1,1,-4
25;-3,4,-4,2,-4,2
26;-3,3,4,-2,3,-1
27;-1,2,-3,-4,-4,4
0;0,0,0,0,0,0
```

9.2 Valeurs initiales

```
"""
contient toutes les valeurs pour les initialisations
"""

#fichier contenant les faces des cubes
"""
fichier de donnee de la forme
numero du cube;liste des faces
le cube 0 sert uniquement pour l'initialisation de la figure
"""
nom_fichier_cubes = "cubes"

#variables d'initialisation des objets cubes
position_init = (-1, -1, -1)
angle_init = 1

#variables d'initialisation de la figure a realiser
longueur = 2
largeur = 3
hauteur = 1
```

9.3 Appel des fonctions

```
import algo_genetique as ag
import gestion_cubes as g
import gestion_figure as gf
import mutation as Mut

liste=ag.gen(g.objets_cubes)
```

```
solution=[]
for i in liste:
  gf.afficher_fig(i)
  if not Mut.Pos_libre(i) :
    solution=i

if (solution):
  gf.afficher_fig(solution)
  for i in range(len(solution)):
    for j in range(len(solution[0])):
      for k in range(len(solution[0][0])):
        L_vois=g.MAJ_L_voisins(g.objets_cubes, solution[i][j][k].num, solution)
        print("{0} : {1}" .format(solution[i][j][k].num, solution[i][j][k].L_faces))

else :
  print("pas de solution trouvee")
```

9.4 Algorithme génétique

```
import gestion_cubes as g
import mutation as Mut
import meilleur as Me
import croisement as C
import random as R
import tirage as t
import ordres as o
import gestion_figure as gf

nb_iter_max = 100000

def gen(dico_cube):
  """
  applique l'algo gen a un dico pour construire la figure entree en parametre
  """
  liste_fig=[]
  liste_c= g.liste_cube(dico_cube)

  for i in range(len(liste_c)) :
    liste_fig.append(gf.init_fig(o.hauteur, o.largeur, o.longueur))
    g.placer_cube((0, 0, 0), liste_c[i], liste_fig[i], dico_cube)

  i=0
  while (not(Mut.est_pleine(liste_fig)) and i<nb_iter_max):

    liste_pond=t.ponderation(liste_fig)

    place_p=t.choix_pond(liste_pond)

    porteuse=liste_fig.pop(place_p)

    place_d=t.choix_alea(liste_fig)

    donneuse=liste_fig.pop(place_d)

    porteuse_temp=porteuse
    donneuse_temp=donneuse#matrice 3D


    croismnt=C.croisement(porteuse_temp,donneuse_temp, dico_cube)
    mut= R.randint(1,10)

    if (mut==1 and Me.nb_cube(donneuse_temp)):
      Mut.mutation(porteuse_temp, liste_fig, dico_cube)

    if (Me.nb_cube(porteuse_temp)>=Me.nb_cube(porteuse) and croismnt):
      liste_fig.append(porteuse_temp)
    else:
      liste_fig.append(porteuse)
```

```
    if (Me.nb_cube(donneuse_temp)!=0): #on remet la donneuse dans la liste que si elle
                                        est non vide
      liste_fig.append(donneuse_temp)
    else:
      liste_fig.append(donneuse)

    if (len(liste_fig)<=2):
      for k in range(len(liste_c)) :
        liste_fig.append(gf.init_fig(o.hauteur, o.largeur, o.longueur))
        g.placer_cube((0, 0, 0), liste_c[k], liste_fig[k], dico_cube)

    liste_fig=gf.en_double(liste_fig)
    liste_fig=gf.doublon(liste_fig)
    i+=1

  return liste_fig
```

9.5 Définition des cubes

```
"""
contient la definition de l'objet cube et toutes les fonctions associees.
"""

import sys

class Cube () :
    def __init__(self, position, L_faces, numero, angle):
  """
    position : tuple (triplet) correspondant a la position (X, Y, Z) dans la forme a
                                                    obtenir
  orientation :  identifiant de la face tournee vers le bas
  L_faces :  liste de la position de chaque face (forme)
  numero :  identifiant du cube
  L_voisins :  liste des voisins du cube
  angle :  forme du bas 'horizontale' ou 'verticale'
  """
        self.X =    position[0]
        self.Y =    position[1]
        self.Z =    position[2]
        self.L_faces =  L_faces
        self.num =    numero
  self.angle =    angle

    def rotation(self, axe, nb_rot):
  """
  definit le mouvement des face lors de la rotation du cube selon les axes X, Y ou Z
                                            dans le sens direct; gere egalement la
                                            modification de l'angle de la forme
  nb_rot :  nombre de rotations de 90 deg a faire : 1 = 1/4 de tour, 2 = 1/2 tour, 3 =
                                            -1/4 de tour
  axe :     axe autour duquel aura lieu la rotation : "X", "Y", ou "Z"
  """
        if axe == "X" :
            for i in range (nb_rot):
                temp=self.L_faces[0] #permutation circulaire
                self.L_faces[0] = self.L_faces[1]
                self.L_faces[1] = self.L_faces[5]
                self.L_faces[5] = self.L_faces[3]
                self.L_faces[3] = temp
    #2 et 4 ne font que tourner
                self.angle*=-1


        elif axe == "Y":
            for i in range (nb_rot):
                temp=self.L_faces[1] #permutation circulaire
                self.L_faces[1] = self.L_faces[4]
                self.L_faces[4] = self.L_faces[3]
```

```
                self.L_faces[3] = self.L_faces[2]
                self.L_faces[2] = temp
    #0 et 5 ne font que tourner
                self.angle*=-1

        elif axe == "Z":
            for i in range (nb_rot):
                temp=self.L_faces[0] #permutation circulaire
                self.L_faces[0] = self.L_faces[4]
                self.L_faces[4] = self.L_faces[5]
                self.L_faces[5] = self.L_faces[2]
                self.L_faces[2] = temp
    #1 et 3 ne font que tourner
                self.angle*=-1



    def deplacement(self, axe, nb_depl, longueur, largeur, hauteur):
    """
    definit la modification de la position d'un cube lors de son deplacement selon X, Y
                                            ou Z; gere egalement les eventuelles
                                            sorties du cube
    nb_depl :   nombre de deplacement de 1 souhaite
    longueur :  longueur de la forme a obtenir, dimension selon X, a recuperer dans la
                                            figure
    largeur :   largeur de la forme a obtenir, dimension selon Y, a recuperer dans la
                                            figure
    hauteur :   hauteur de la forme a obtenir, dimension selon Z, a recuperer dans la
                                            figure
    """
        if axe == "X":
            self.X += nb_depl
            if self.X > longueur : sys.exit ("X hors cube")

        elif axe == "Y":
            self.Y += nb_depl
            if self.Y > largeur : sys.exit ("Y hors cube")

        elif axe == "Z":
            self.Z += nb_depl
            if self.Z > hauteur : sys.exit ("Z hors cube")
```

9.6 Fonctions de gestion des cubes

```
"""
contient l'initialisation de la figure a realiser et les fonctions qui permettent de
                                            placer des cube dans celle-ci
"""


import definition_cube as d
import ordres as o
import gestion_figure as gf

#----dictionnaire des faces des cubes

fichier = open(o.nom_fichier_cubes, 'r')

"""
creation d'un dictionnaire contenant toutes les faces des cubes repertories dans le
                                            fichier ouvert sous la forme num: liste des
                                             faces
faces_cubes :   dictionnaire des cubes et de leurs faces
C :     variable temporaire de recuperation des donnees sous forme de chaine de
                                            caracteres
"""

faces_cubes = {}
for line in fichier:
```

```
  C=line.split(";")
  faces_cubes[int(C[0])] = [int(i) for i in C[1].split(",")]

fichier.close()

#----dictionnaire des objets cubes

"""
creation dun dictionnaire contenant les objets cubes a partir du dictionnaire des
                                        faces
position_init :   position initiale du cube, a l'exterieur de la figure
angle_init :    angle initial du cube
L_voisins_init :  liste initiale des voisins du cube, initialisee a 0, de la forme [Y-
                                        1, Z+1, X+1, Z-1, X-1, Y+1]
objets_cubes :    dictionnaire des objets cubes avec leurs carateristiques (.X, .Y, .Z
                                        , .orientation, .L_faces, .num, .L_voisins,
                                         .angle)
"""

objets_cubes = {}
for i in faces_cubes:
  objets_cubes[i] = d.Cube(o.position_init, faces_cubes[i], i, o.angle_init)


#----verification de la compatibilitee de deux cubes

def compatibilite(cube_1, cube_2, dico_cube):
  """
  retourne True si un cube peu bien etre place a l'endroit designe et False sinon
  compare les faces en contact (forme, complementaritee) et verifie que les cubes sont
                                        sous le meme angle entre un cube et chacun
                                        de ses voisins
  cube_1 :  numero du premier cube a comparer, int
  cube_2 :  numero du deuxieme cube a comparer, int
  dico_cube : dictionnaire d'objets cubes
  ok :    booleen verifiant si les cubes sont compatibles ou non
  """
  ok=False

  if (dico_cube[cube_1].angle == dico_cube[cube_2].angle) :
    if dico_cube[cube_1].X < dico_cube[cube_2].X :
      ok = (dico_cube[cube_1].L_faces[2] + dico_cube[cube_2].L_faces[4] == 0)

    elif dico_cube[cube_1].X > dico_cube[cube_2].X :
      ok = (dico_cube[cube_1].L_faces[4] + dico_cube[cube_2].L_faces[2] == 0)

    elif dico_cube[cube_1].Y < dico_cube[cube_2].Y :
      ok = (dico_cube[cube_1].L_faces[5] + dico_cube[cube_2].L_faces[0] == 0)

    elif dico_cube[cube_1].Y > dico_cube[cube_2].Y :
      ok = (dico_cube[cube_1].L_faces[0] + dico_cube[cube_2].L_faces[5] == 0)

    elif dico_cube[cube_1].Z < dico_cube[cube_2].Z :
      ok = (dico_cube[cube_1].L_faces[1] + dico_cube[cube_2].L_faces[3] == 0)

    elif dico_cube[cube_1].Z > dico_cube[cube_2].Z :
      ok = (dico_cube[cube_1].L_faces[3] + dico_cube[cube_2].L_faces[1] == 0)

  return ok

#----mise a jour de la liste des voisins

def MAJ_L_voisins(dico_cube, cube, figure):
  """
  met a jour la liste des voisin du cube numero cube dans le dictionnaire dico_cube
  dico_cube : dictionnaire d'objets
  cube :    numero du cube, int
  figure :  figure a realiser
  """
  #si la coordonnee que l'on verifie est dans la figure et que le contenu de la figure
                                        en cette coordonnee est different de 0
```

```
                                        alors :
    #ajouter le numero du cube qui s'y trouve a la liste des voisins du cube par
                                        rapport auquel on travaille


  L_voisins=[0 for i in range(6)]

  if dico_cube[cube].X + 1 < len(figure) and figure[dico_cube[cube].X + 1][dico_cube[
                                        cube].Y][dico_cube[cube].Z].num != 0 :
    L_voisins[2] = figure[dico_cube[cube].X + 1][dico_cube[cube].Y][dico_cube[cube].Z]
                                        .num

  if dico_cube[cube].Y + 1 < len(figure[0]) and figure[dico_cube[cube].X][dico_cube[
                                        cube].Y + 1][dico_cube[cube].Z].num != 0 :
    L_voisins[5] = figure[dico_cube[cube].X][dico_cube[cube].Y + 1][dico_cube[cube].Z]
                                        .num

  if dico_cube[cube].Z + 1 < len(figure[0][0]) and figure[dico_cube[cube].X][dico_cube
                                        [cube].Y][dico_cube[cube].Z + 1].num != 0 :
    L_voisins[1] = figure[dico_cube[cube].X][dico_cube[cube].Y][dico_cube[cube].Z + 1]
                                        .num

  if dico_cube[cube].X - 1 >= 0 and figure[dico_cube[cube].X - 1][dico_cube[cube].Y][
                                        dico_cube[cube].Z].num != 0 :
    L_voisins[4] = figure[dico_cube[cube].X - 1][dico_cube[cube].Y][dico_cube[cube].Z]
                                        .num

  if dico_cube[cube].Y - 1 >= 0 and figure[dico_cube[cube].X][dico_cube[cube].Y - 1][
                                        dico_cube[cube].Z].num != 0 :
    L_voisins[0] = figure[dico_cube[cube].X][dico_cube[cube].Y - 1][dico_cube[cube].Z]
                                        .num

  if dico_cube[cube].Z - 1 >= 0 and figure[dico_cube[cube].X][dico_cube[cube].Y][
                                        dico_cube[cube].Z - 1].num != 0 :
    L_voisins[3] = figure[dico_cube[cube].X][dico_cube[cube].Y][dico_cube[cube].Z - 1]
                                        .num

  return L_voisins


#----demande de placement

def placement_possible(position, cube, figure, dico_cube):
  """
  retourne True si le placement est possible et False sinon
  position :  coordonnees ou l'on veut placer le cube, triplet
  cube :    numero du cube a placer, int
  figure :  figure a obtenir, matrice 3D
  dico_cube : dictionnaire d'objets cubes
  ok :    booleen verifiant si le placement est possible ou non
  """
  L_voisins=MAJ_L_voisins(dico_cube, cube, figure)
  ok = False
  i = 0

  if (L_voisins==[0 for i in range(6)]) :
    ok=True
  else :
    while not(ok) and i< len(L_voisins) :
      if L_voisins[i] != 0 and not(ok):
        ok = compatibilite(dico_cube[cube].num, dico_cube[i].num, dico_cube)
      i+=1
  return ok



#----placement d'un cube dans la figure

def placer_cube(position, cube, figure, dico_cube):
  """
```

```
  place un cube dans la matrice de la forme et met a jour la liste des voisins de ce
                                                cube et de ceux autour
  position :  coordonnees ou l'on veut placer le cube dans la figure, triplet
  cube :    numero du cube a placer, int
  figure :  figure a obtenir, matrice 3D
  dico_cube : dicionnaire d'objets cubes
  """
  place=False
  x=position[0]
  y=position[1]
  z=position[2]
  if (figure[x][y][z].num==0):
    if placement_possible(position, cube, figure, dico_cube) and not gf.present(cube,
                                                figure):
      if x <len(figure) and y <len(figure[0]) and z <len(figure[0][0]) and x >=0 and y
                                                 >=0 and z >=0 :
        figure[x][y][z] = dico_cube[cube]

        dico_cube[cube].X = x
        dico_cube[cube].Y = y
        dico_cube[cube].Z = z
        place=True

        L_voisins=MAJ_L_voisins(dico_cube, cube, figure)

  return place

#----retirer un cube de la figure

def retirer_cube_C(cube, figure, dico_cube):
  """
  retire un cube de la matrice de la forme a partir du numero du cube et met a jour la
                                                 liste des voisins de ce cube et de ceux
                                                autour
  cube :    numero du cube a placer, int
  figure :  figure a obtenir, matrice 3D
  dico_cube : dicionnaire d'objets cubes
  """
  figure[dico_cube[cube].X][dico_cube[cube].Y][dico_cube[cube].Z] = dico_cube[0]

  dico_cube[cube].X = -1
  dico_cube[cube].Y = -1
  dico_cube[cube].Z = -1

  L_voisins=MAJ_L_voisins(dico_cube, cube, figure)



def retirer_cube_P(position, figure, dico_cube):
  """
  retire un cube de la matrice de la forme a partir d'une position et met a jour la
                                                liste des voisins de ce cube et de ceux
                                                autour
  position :  coordonnees ou l'on veut placer le cube dans la figure, triplet
  figure :  figure a obtenir, matrice 3D
  dico_cube : dicionnaire d'objets cubes
  """
  if position[0] < len(figure) and position[1] < len(figure[0]) and position[2] < len(
                                                figure[0][0]) and position[0] >= 0 and
                                                position[1] >= 0 and position[2] >= 0 :

    cube = figure[position[0]][position[1]][position[2]].num
    figure[position[0]][position[1]][position[2]] = dico_cube[0]

    dico_cube[cube].X = -1
    dico_cube[cube].Y = -1
    dico_cube[cube].Z = -1

    L_voisins=MAJ_L_voisins(dico_cube, cube, figure)

  return cube
```

```
#----liste des cubes

def liste_cube(dico_cube):
  """
  renvoie la liste des numeros des cubes du dictionnaire
  dico_cube : dicionnaire d'objets cubes
  """
  liste_c=[]
  for k in dico_cube :
    liste_c.append(k)
  liste_c.remove(0)
  return (liste_c)
```

9.7 Fonctions de gestion des figures

```
import gestion_cubes as g

#----initialisation de la figure

"""
creation de la figure que l'on souhaite obtenir, initialise a 0 la matrice 3D de la
                                                    forme a realiser
forme de type parallellepipede rectangle : longeur, largeur, hauteur
longueur :  longeur de la forme a obtenir, dimension selon X
largeur :   largeur de la forme a obtenir, dimension selon Y
hauteur :   hauteur de la forme a obtenir, dimension selon Z

"""
def init_fig(hauteur, largeur, longueur):
  return [[[g.objets_cubes[0] for k in range(hauteur)] for j in range(largeur)]for i
                                                    in range(longueur)]
#print (figure)


#----cube deja dans la figure ?

def present(cube, figure):
  """
  verifie si un cube est present ou non dans la figure
  cube :    numero du cube a placer, int
  figure :  figure a obtenir, matrice 3D
  present : teste si le cube est present, booleen
  """
  present = False
  i = 0
  while not present and i<len(figure):
    present=cube in figure[i]
    i+=1
  return present

#----affichage de la figure

def afficher_fig(figure):
  """
  affiche les numeros des cubes present dans la figure donnee en argument
  figure :  figure a obtenir, matrice 3D
  """
  R = [[[figure[i][j][k].num for k in range (len(figure[0][0]))] for j in range (len(
                                                    figure[0]))] for i in range (len(figure))]
  print(R)

#---liste des cubes dans la figure

def liste_cube_fig(figure):
  cube=[]
  for i in range (len(figure)):
    for j in range (len(figure[0])):
```

```
      for k in range (len(figure[0][0])):
        if figure[i][j][k].num!=0 :
          cube.append(figure[i][j][k].num)
  return cube


#----liste des cubes pas encore utilises

def Cubes_libre_fig(figure):
  """
  constitution de la liste des cubes encore disponibles en vue de les placer
  figure :  figure a obtenir, matrice 3D
  cubes_libre : liste des cubes encore disponibles par rapport a la figure donnee
  """
  cubes_libres=[]
  for i in range (1,28) :
    if present(i, figure):
      cubes_libres.append(i)
  return cubes_libres


#---suppression des figures en double dans la liste des sous solutions

def en_double(liste_fig) :
  for i in liste_fig :
    if liste_fig.count(i)>1 :
      liste_fig.remove(i)
  return(liste_fig)


#----suppression des figures contenant des cubes en double

def doublon(liste_fig):
  for i in liste_fig :
    lst=liste_cube_fig(i)
    dbl=False
    j=0
    while not dbl and j<len(lst) :
      dbl=not(lst.count(lst[j])==1 or lst[j]==0)
      j+=1
    if dbl :
      liste_fig.remove(i)
  return(liste_fig)
```

9.8 Pondération des figures

```
#----combien de cubes dans une figure ?

def nb_cube(figure):
  """
  donne le nombre de places occupees dans une figure
  figure :  figure a obtenir, matrice 3D
  """
  nb = 0
  for i in range (len(figure)):
    for j in range (len(figure[0])):
      for k in range (len(figure[0][0])):
        if figure[i][j][k].num != 0 :
          nb += 1
  return nb
```

9.9 Mutation d'une figure

```
import random as R
import gestion_cubes as g
import gestion_figure as gf
```

```
import ordres as o

#----liste des positions libres

def Pos_libre(figure):
  """
  costitution de la liste des positions encore libres dans la figure
  pos_libres :  liste des positions encore libres dans la figure
  figure :  figure a obtenir, matrice 3D
  """
  pos_libres=[]
  for i in range (len(figure)):
    for j in range (len(figure[0])):
      for k in range (len(figure[0][0])):
        if figure[i][j][k].num==0 :
          pos_libres.append((i, j, k))
  return pos_libres


#---ya il une figure pleine ?

def est_pleine(liste_fig):
  """
  definit si une liste de figures contient une figure pleine
  liste_fig = liste des sous solutions
  """
  plein=False
  cpt=0
  while (not plein and cpt<len(liste_fig)):
    plein=(len(Pos_libre(liste_fig[cpt]))==0)
    cpt+=1
  return plein


#----mutation

def mutation(figure, liste_fig, dico_cube):
  """
  fait une mutation sur la figure
  """

  cube_m=R.choice(gf.liste_cube_fig(figure)) #int
  g.retirer_cube_C(cube_m, figure, dico_cube)
  axe=R.choice(['X','Y','Z'])
  dico_cube[cube_m].rotation(axe,2)

  place=False
  cpt=0
  position=Pos_libre(figure)
  #print(position)
  while (not place and cpt<len(position)):
    place=g.placer_cube(position[cpt], cube_m, figure, dico_cube)
    cpt+=1

  if (not place) :
    liste_fig.append(gf.init_fig(o.hauteur, o.largeur, o.longueur))
    g.placer_cube((0, 0, 0), cube_m, liste_fig[-1], dico_cube)
```

9.10 Croisement de deux figures

```
import gestion_cubes as g
import gestion_figure as gf
import mutation as Mut

#----croisement de deux figures

def croisement(porteuse_temp,donneuse_temp, dico_cube):
  liste_cube_donn = gf.liste_cube_fig(donneuse_temp) #liste_cube_fig
  liste_pos_por = Mut.Pos_libre(porteuse_temp)
```

```
  ajouter=False
  cpt_c=0
  cpt_pos=0
  cpt_rot_x=0
  cpt_rot_y=0
  cpt_rot_z=0

  while (not ajouter and cpt_c<len(liste_cube_donn)):
    cube=dico_cube[liste_cube_donn[cpt_c]]
    while (not ajouter and cpt_pos<len(liste_pos_por)):
      while (not ajouter and cpt_rot_x<3):
        while (not ajouter and cpt_rot_y<3):
          while (not ajouter and cpt_rot_z<3):
            ajouter=g.placer_cube(liste_pos_por[cpt_pos], liste_cube_donn[cpt_c],
                                  porteuse_temp, dico_cube)
            cube.rotation('Z', 1)
            cpt_rot_z+=1
          cube.rotation('Y', 1)
          cpt_rot_y+=1
        cube.rotation('X', 1)
        cpt_rot_x+=1
      cpt_pos+=1
    cpt_c+=1
  return ajouter #croisement fait ou non ?
```

9.11 Tirage pondéré

```
import random as R
import meilleur as Me
import gestion_cubes as g

#----ponderation de la liste des figures

def ponderation(liste_fig):
  """
  retourne la liste ponderee et normee des sous solutions
  """
  liste_pond=[]
  for i in range (len(liste_fig)):
    liste_pond.append(float(Me.nb_cube(liste_fig[i]))/float(len(g.objets_cubes)))
  return liste_pond

#----choix pondere d'une figure

def choix_pond(liste_pond):
  """
  choix stochastique de la figure "porteuse"
  nb est initialise a 1-rnd car random() prends ses valeurs sur [0,1[ et on veut nos
                                        valeurs sur ]0,1]
  ceci ne change rien a la modelisation si on assimile les float a un interval continu
  """
  i = 0
  cpt = 0
  nb = 1-R.random()
  while (nb>cpt and i<len(liste_pond)):
    cpt += liste_pond[i]
    i += 1
  return i-1

#----choix aleatoire d'une figure

def choix_alea(liste_fig):
  """
  choix aleatoire de la figure "donneuse"
  """
  nb = R.randint(0,len(liste_fig))-1
  return nb
```

Références

[1] *Was Darwin a Great Computer Scientist ?*
Louis Nicolle
https://blog.sicara.com/getting-started-genetic-algorithms-python-tutorial-81ffa1dd72f9

[2] *Genetic Algorithm*
page Wikipédia
https://en.wikipedia.org/wiki/Genetic_algorithm

[3] *Introduction to Genetic Algorithms-Including example Code*
Vijini Mallawaarachchi
https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3