Application de méthodes de recherche d'isomorphismes de graphes à la comparaison des structures de protéines

Adrien Dubois - N°candidat : 51771

5 juin 2022

Plan

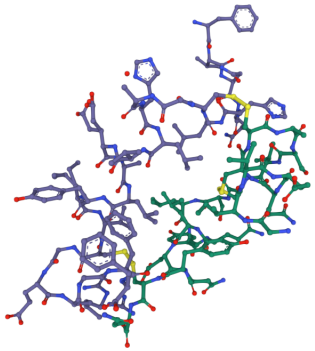
Contexte, données à traiter et objectif

Compétition Casp

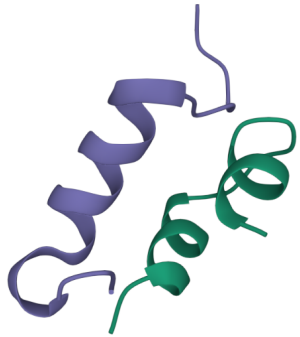
Sequence of 1PID | BOVINE...

```
1          11          21
GIVEQCCASVCSLYQLENYCN
```

Séquence



Structure



Représentation simplifiée

ex : protéine d'insuline bovine [1]

Objectif : proposer une méthode de comparaison de structures

[1] source : Protein Data Bank, https ://www.rcsb.org/3d-view/1B0Q

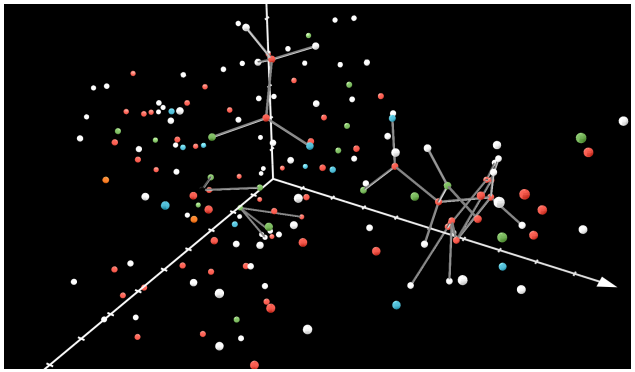
Format PDB et représentation

```
ATOM    171 HG13 VAL A  10      -7.189  -6.908  -5.228  1.00  0.00           H
HETATM  172  N   NH2 A  11      -3.913  -3.201  -4.868  1.00  0.00           N
HETATM  173  HN1 NH2 A  11      -3.068  -3.568  -5.283  1.00  0.00           H
HETATM  174  HN2 NH2 A  11      -3.878  -2.361  -4.308  1.00  0.00           H
TER     175      NH2 A  11
HETATM  176 RE    RE A 182       2.230  -1.164   0.585  1.00  0.00          RE
CONECT    1    2    3    7
CONECT    2    1
CONECT    3    1    4    5    6
```

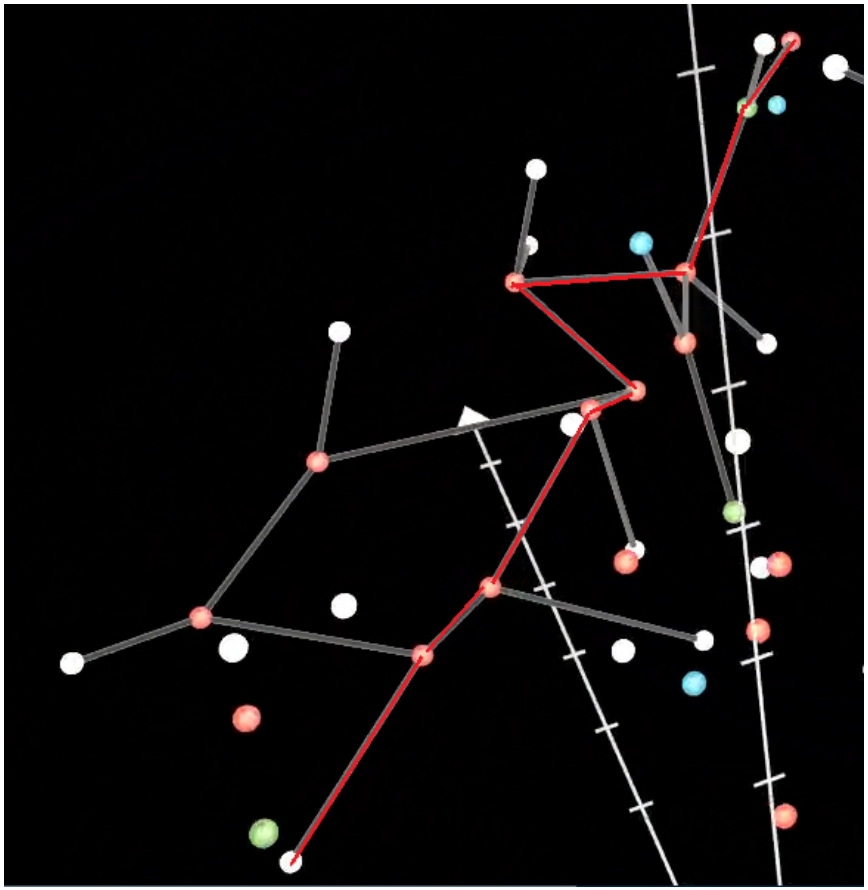
FIG. : Lignes d'un fichier PDB

Données

position des atomes	→	OK
type des atomes	→	OK
liaisons	→	moyennes



Branche sans ramification



en python :

- type : Coord : *float* $*$ *float* $*$ *float*
- type : Branche : liste de points
  ex : $B = [(0,0,0),(1,1,1),(1.5,1,2)]$

Comparaison de branches : distances

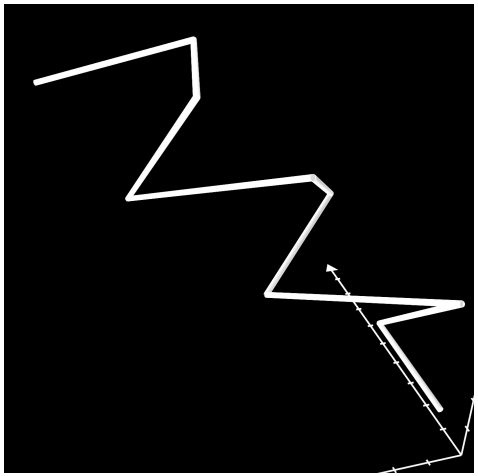


FIG. : Branche A

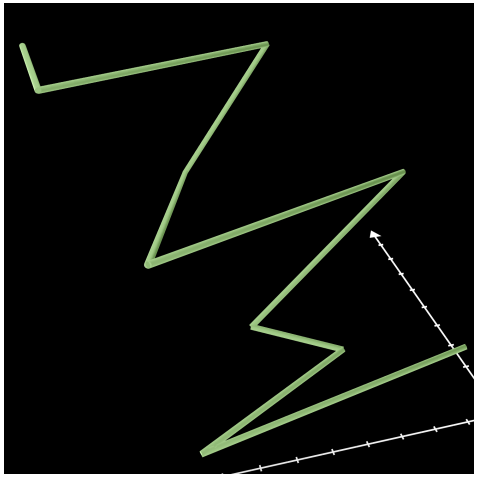
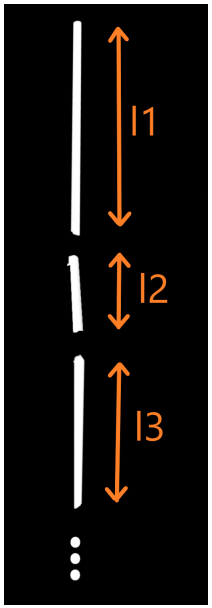
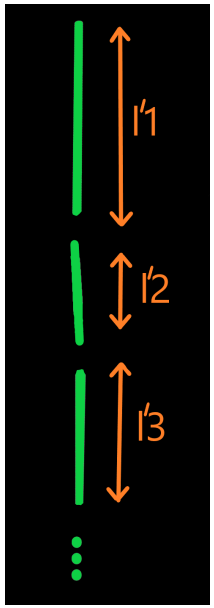


FIG. : Branche B





Comparaison de branches : angles

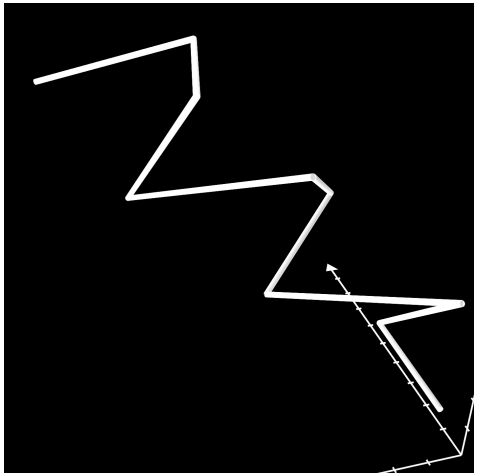


FIG. : Branche A

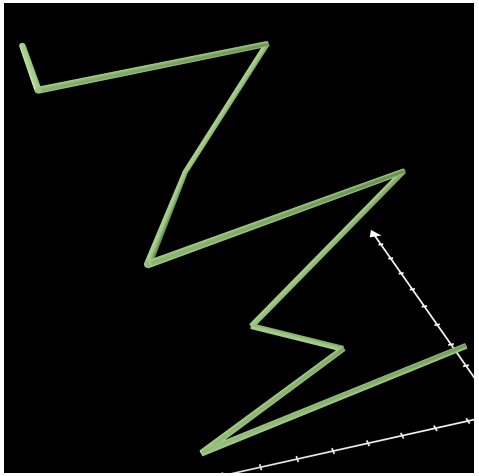


FIG. : Branche B

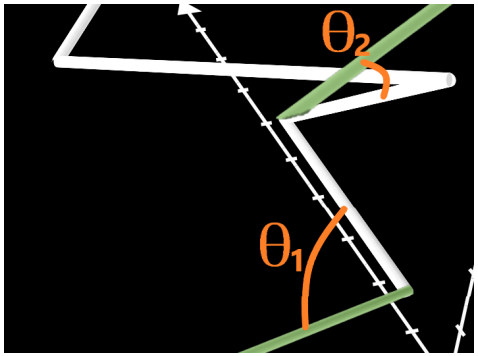


FIG. : Angles

Coefficient de proximité de branches

$$\forall i \in [\![1, n-1]\!],\ d_i = max(l_i, l'_i)$$

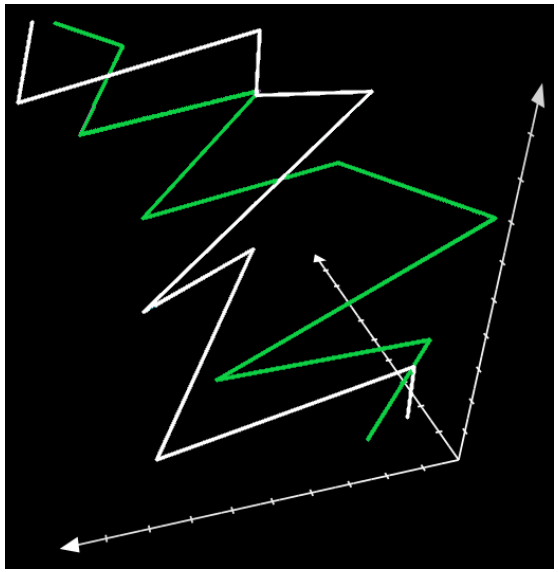
$$C_{dist} = \frac{\sum_{i=0}^{n-1} |\sin(\theta_i)| d_i}{\sum_{i=0}^{n-1} d_i} \qquad C_{angle} = \frac{\sum_{i=0}^{n-1} |l'_i - l_i|}{\sum_{i=0}^{n-1} d_i}$$

$$R_{dist} = \frac{1}{1 + C_{dist}} \qquad R_{angle} = \frac{1}{1 + C_{angle}}$$

Finalement, on définit :

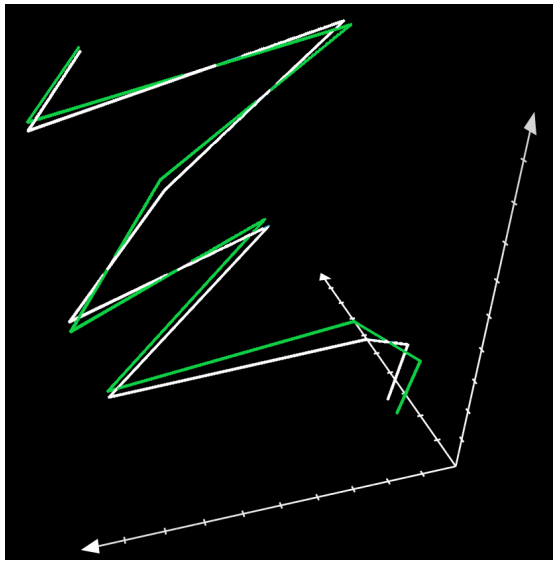
$$\boxed{R = \frac{R_{dist} + R_{angle}}{2}}$$

Valeur du coefficient sur exemples



$$R_{dist} = 0.59 \quad R_{angle} = 0.68$$

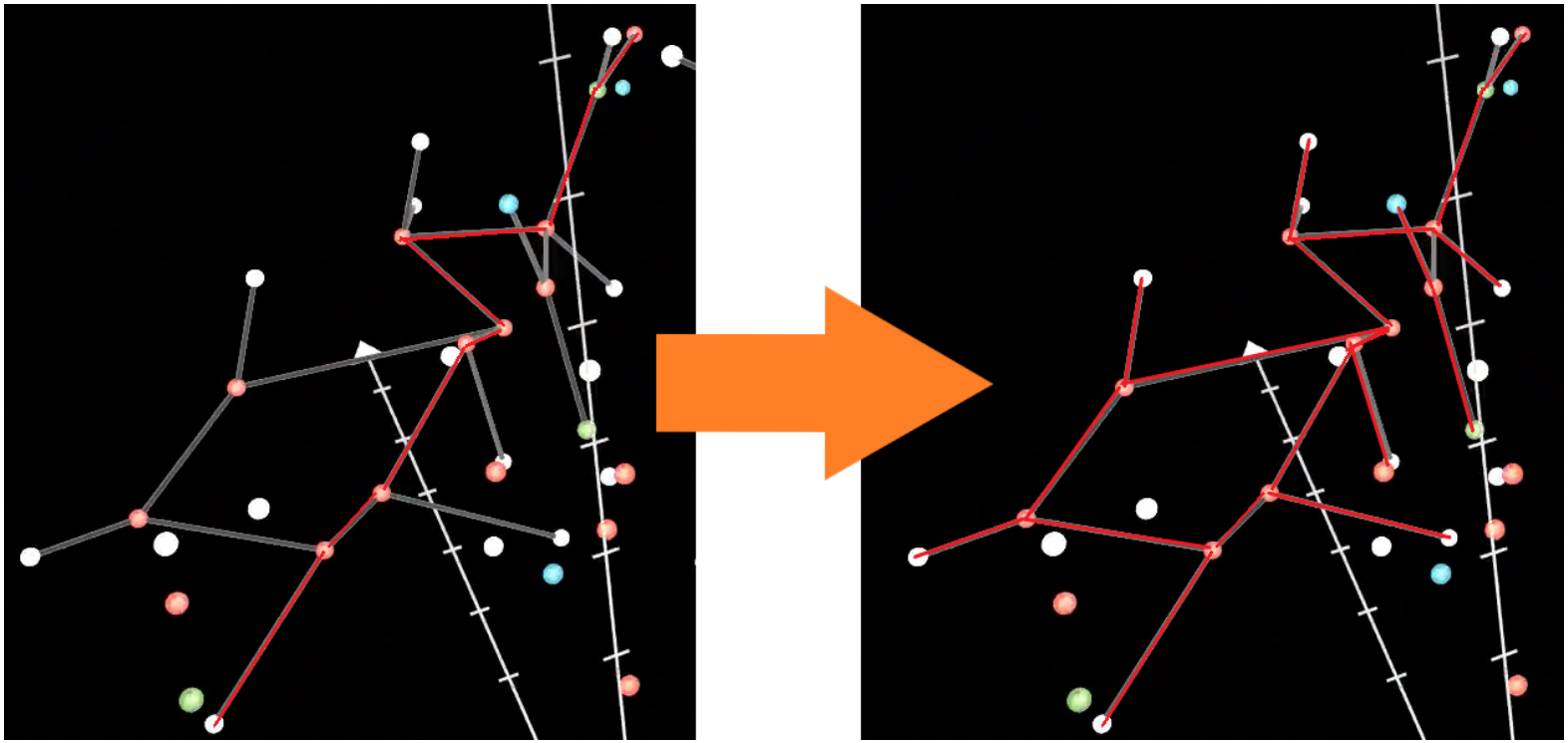
$$\boxed{R = 0.64}$$



$$R_{dist} = 0.94 \quad R_{angle} = 0.97$$

$$\boxed{R = 0.95}$$

Des branches à la protéine complète



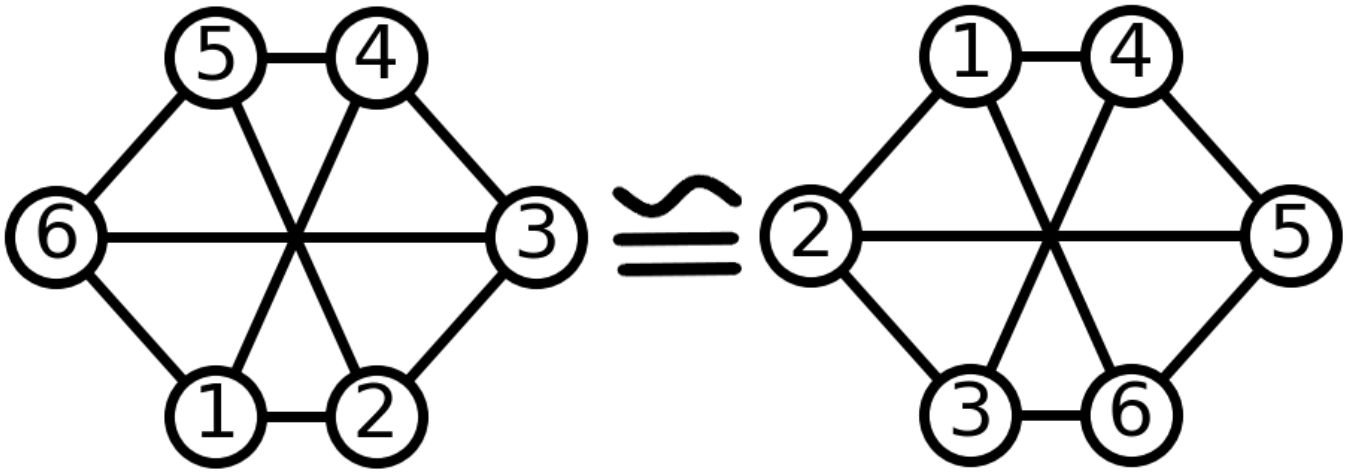
Définition d'un isomorphisme de graphes

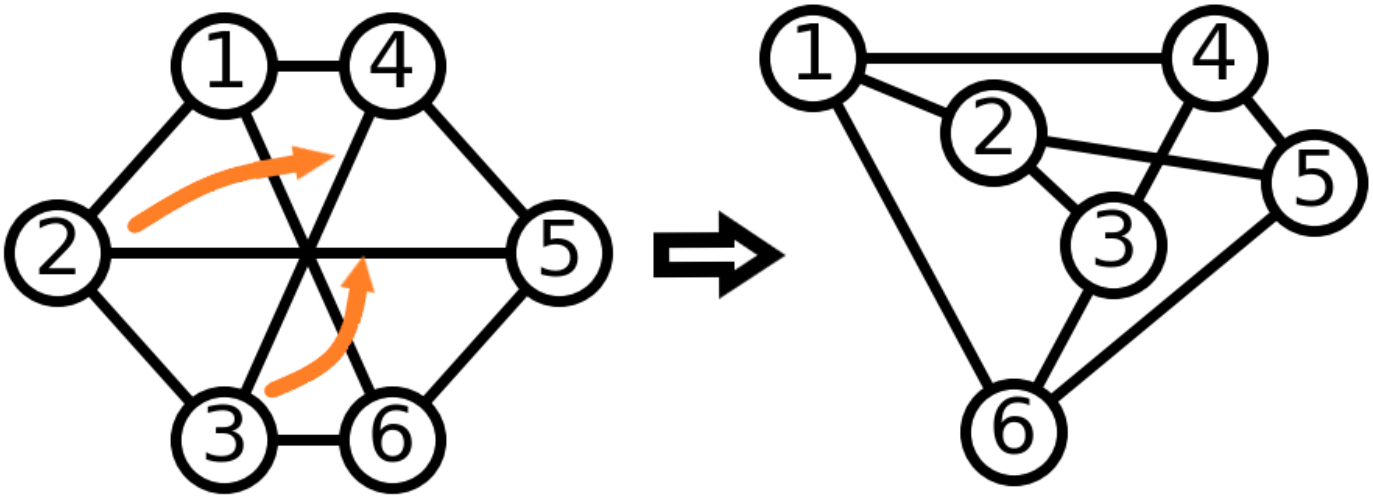
Définiton : Soit G=$A_g \times S_g$ et H=$A_h \times S_h$ deux graphes où $S_g, S_h \subseteq [\![1, n]\!]^2$ (l'arête (g_i, g_j) est dans G $\iff$ $(i, j) \in S_g$) :

$$G \cong H \text{ si et seulement si } \exists \sigma \in \Sigma_n, G = H^\sigma \quad (1)$$

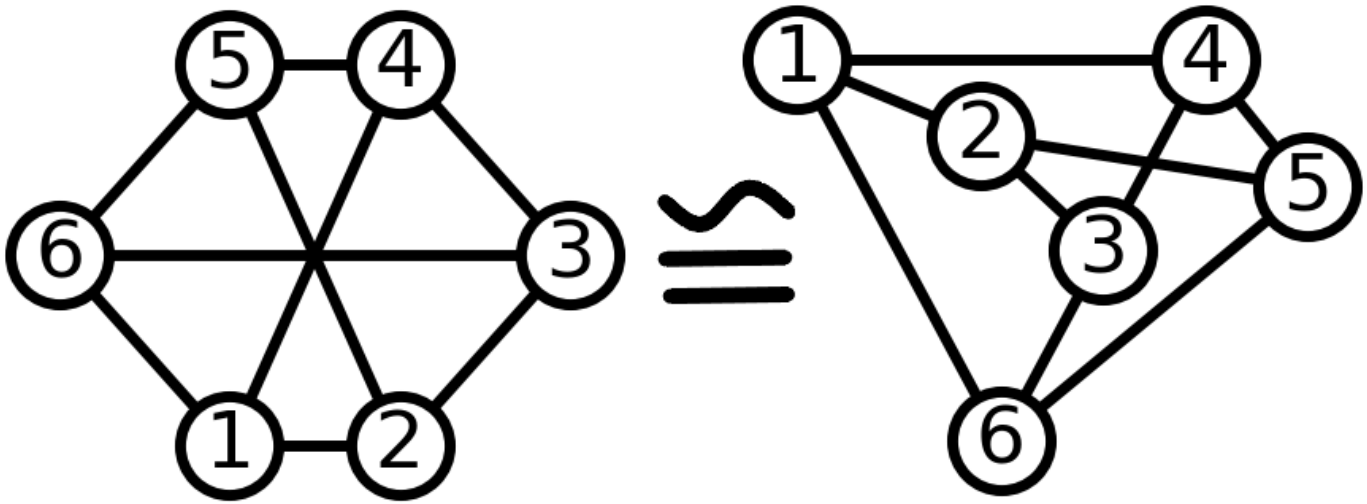
où $H^\sigma := A_h \times \{(\sigma(i),\ \sigma(j)),\ \exists (i,j) \in [\![1, n]\!]^2,\ (i,j) \in S_h\}$

Exemple :

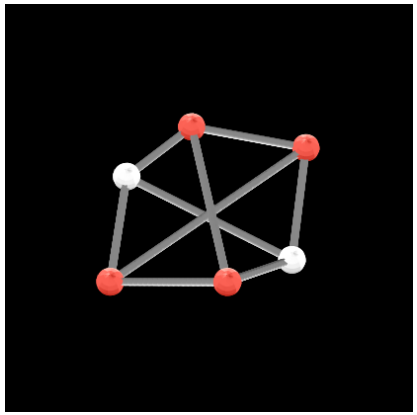


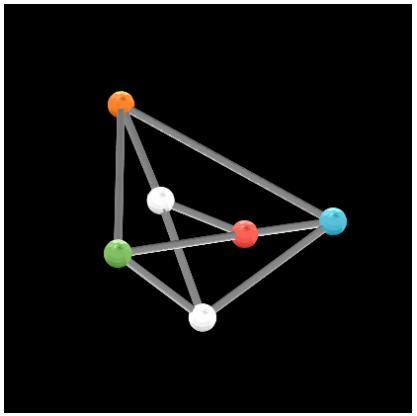


donc :



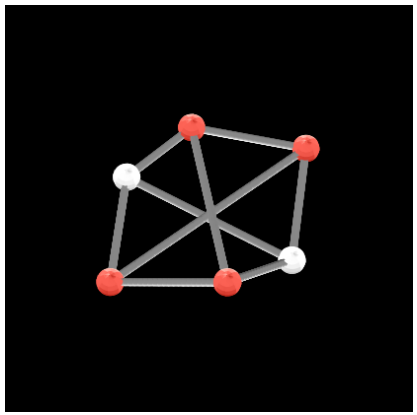
Notion d'isomorphisme appliquée aux protéines

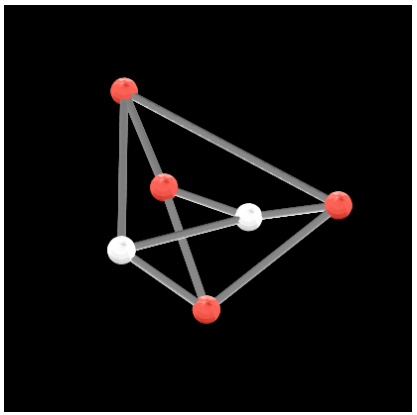




- Même structure
- atomes différents
- positions des atomes différentes

$\rightarrow$ **non isomorphes**





- Même structure
- mêmes atomes
- positions des atomes différentes

$\rightarrow$ **isomorphes**

Idée 1 : Recherche par force brute

On cherche pour $\sigma \in \Sigma_n$, une permutation σ telle que $G = H^\sigma$,
En python :

- type : Graph
  ex : $G_1 = Graph(\ [0,1,2],\ [[0,1],[0],[0]]\ )$
  $H_1 = Graph(\ [0,1,2],\ [[1,0],[2],[2]])$
- type : permutation : int list
  ex : sur Σ_3, $Id = [0,1,2]$, $\sigma_1 = [2,1,0]$ la transpositon de 1 et 3
- fonction : test_isomorphism : Graph, Graph, int list $\longrightarrow$ bool
  ex : $test\_isomorphism(G_1, H_1, [0,1,2])$
  teste si G et H^σ sont les mêmes graphes
- fonction : isomorphes : Graph, Graph $\longrightarrow$ bool
  ex : $isomorphes(G, H) = True$
  G_1 et H_1 sont isomorphes car $\exists \sigma \in \Sigma_3,\ (\sigma = \sigma_1)\ G_1 = H_1^\sigma$

Idée 2 : tri des sommets par invariants

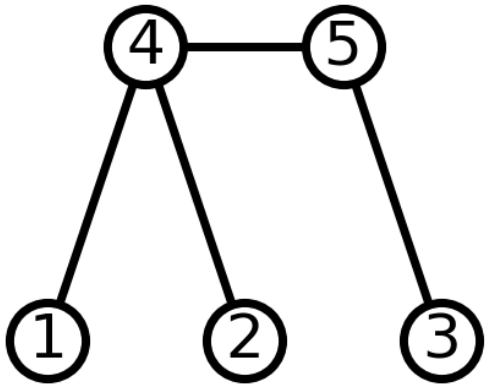
Les invariants des sommets que l'on peut utiliser sont par exemple le degré, le type du sommet,...

Exemple : $G = [\![1, 5]\!] \times S_g$

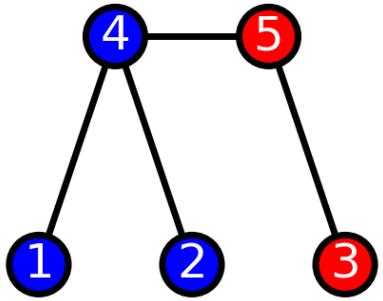
Les sommets ont une couleur : $1,\ 2$ et 4 sont bleus

3 et 5 sont rouges

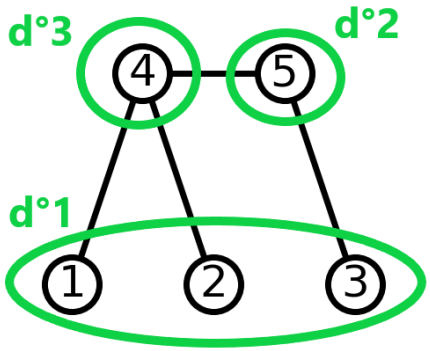
Et ont aussi un degré égal au nombre de voisins



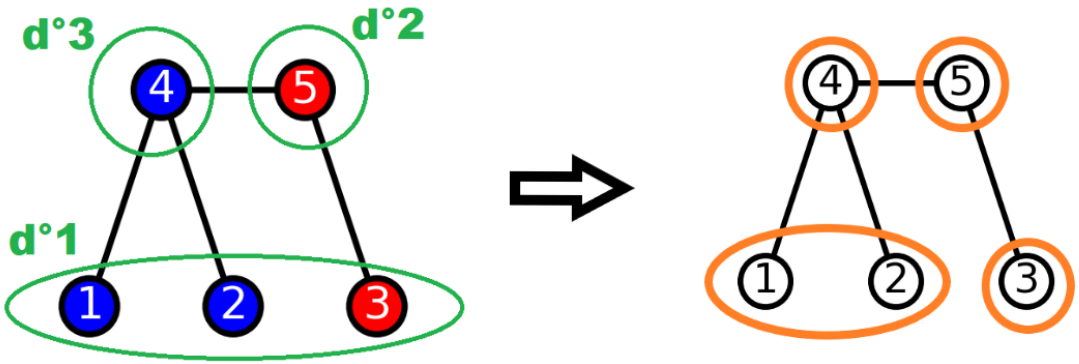
On peut trier les sommets par couleurs : $(1\ 2\ 4 \mid 3\ 5)$



Ou trier les sommets par degré : $(1\ 2\ 3 \mid 5 \mid 4)$

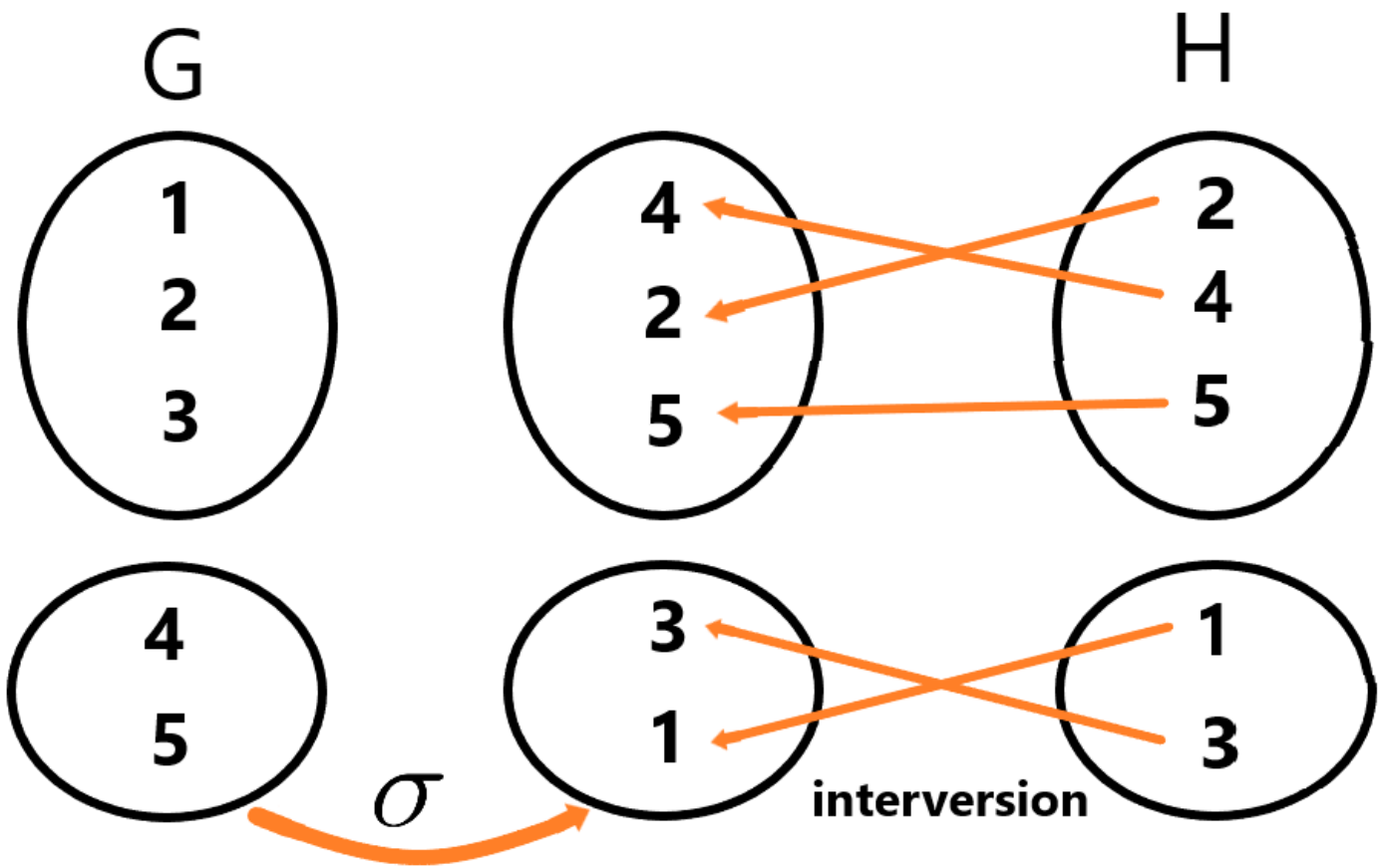


Puis combiner les deux : $(1\ 2 \mid 3 \mid 5 \mid 4)$



Idée 2 : tri des sommets par invariants

Principe de recherche d'isomorphisme avec les sommets triés :



Idée 2 : tri des sommets par invariants

On crée ainsi une partition de $[\![1, n]\!]$ des sommets $(V_1|\dots|V_t)$ pour G et $(W_1|\dots|W_t)$ pour H alors :

$$G \cong H \iff \exists(\sigma_i)_{i\in[\![1,t]\!]} \in \Sigma_{|V_1|} \times .. \times \Sigma_{|V_t|},\ S_g = S_h^{\tilde{\sigma_1}\circ..\circ\tilde{\sigma_t}} \qquad (2)$$

$$\text{avec } \forall v_{i,j} \in V_i,\ \tilde{\sigma_i}(v_{i,j}) = w_{i,\sigma_i(j)} \text{ avec } V_i = (v_{i,1}\ \dots\ v_{i,|V_i|})$$
$$\forall v \notin V_i,\ \tilde{\sigma_i}(v) = v$$

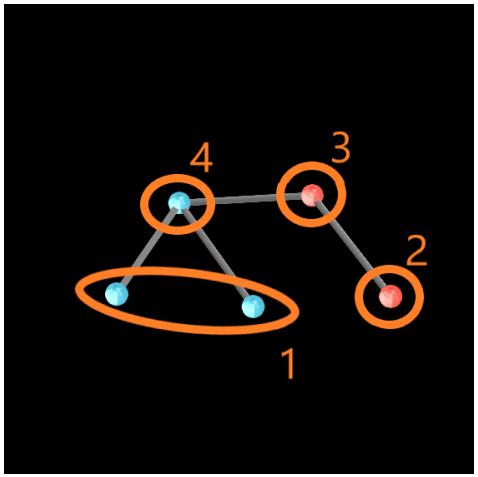
Exemple :

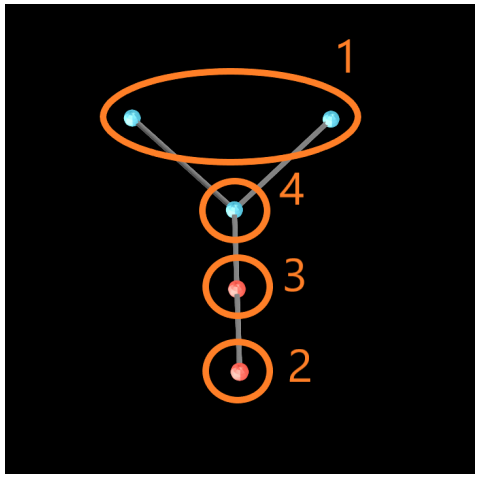
$$(V_1, V_2) \to (\ 1\ \ 2\ \ 3\ \mid\ 4\ \ 5\ )$$
$$(W_1, W_2) \to (\ 1\ \ 3\ \ 5\ \mid\ 2\ \ 4\ )$$

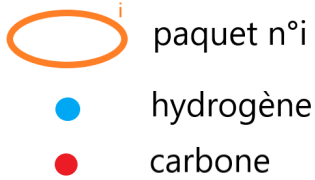
σ_1 [2]	σ_2	$(W_1^{\sigma_1}\|W_2^{\sigma_2})$	$\sigma = \tilde{\sigma_1}\circ\tilde{\sigma_2}$
$(1\ 2\ 3) = Id$	$(1\ 2) = Id$	$(1\ \ 3\ \ 5 \mid 2\ \ 4)$	$(1\ \ 3\ \ 5\ \ 2\ \ 4)$
$(3\ 2\ 1)$	$(1\ 2) = Id$	$(5\ \ 3\ \ 1 \mid 2\ \ 4)$	$(5\ \ 3\ \ 1\ \ 2\ \ 4)$
$(1\ 2\ 3) = Id$	$(2\ 1)$	$(1\ \ 3\ \ 5 \mid 4\ \ 2)$	$(1\ \ 3\ \ 5\ \ 4\ \ 2)$

[2] *pour* $\sigma \in \Sigma_n$, *on note* $\sigma = (\sigma(1)\ \sigma(2)\ \cdots\ \sigma(n))$

Tri sur les protéines







	force brute	tri
permutations de sommets	$5! = 120$	2

Comparaison des complexités force brute / tri

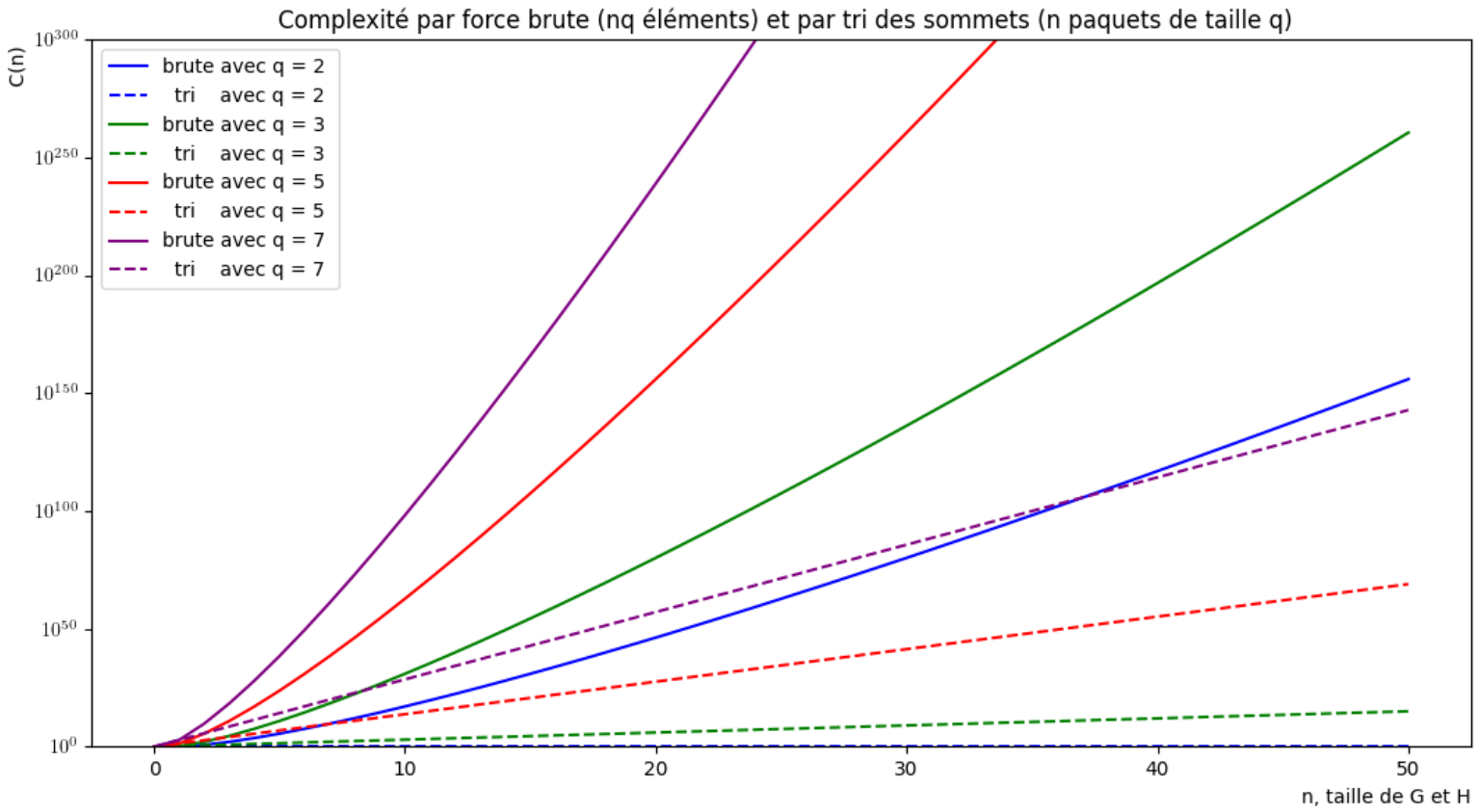
G et H sont des graphes à n sommets tous deux **triés canoniquement** :

- types des $t \in \mathbb{N}$ paquets du tri sont dans le **même ordre**
- on suppose les paquets de **même tailles** $t_i,\ i \in [\![1, t]\!]$ (sinon les graphes seraient trivialement non isomorphes)

	force brute	tri des sommets
procédé d'itération	permutations sur Σ_n	combinaison de permutations sur $(\Sigma_i)_{i \in [\![1,t]\!]}$ tel que $\sum_{i=1}^{t} t_i = n$
nombre maximal d'itérations	$n!$	$\prod_{i=1}^{t} t_i!$
mise en mémoire nécéssaire	$\sum_{k=1}^{n} k!$ permutations	$\sum_{k=1}^{max(t_1,..,t_t)} k!$ permutations

Comparaison des complexités force brute / tri

On considère deux graphes triés en n paquets de taille q :



Idée 3 : Isomorphisme cannonique de McKay

Idée générale :

- tri encore plus fin : par propagation du degré
- création d'un arbre de recherche de permutations
- isomorphisme cannonique

Tri des sommets encore plus fin

Utilisation de la propagation du degré :

Tri des sommets des paquets par degré dans les autres paquets pour en faire un nouveau tri
→ jusqu'à ce qu'il n'y ait plus de simplifications possibles

Exemple :

Soit $\pi = (1 \mid 3\ 7\ 9 \mid 6\ 8 \mid 2\ 4 \mid 5)$ un tri des sommets du graphe G
$= (V_1 \mid V_2 \mid V_3 \mid V_4 \mid V_5)$

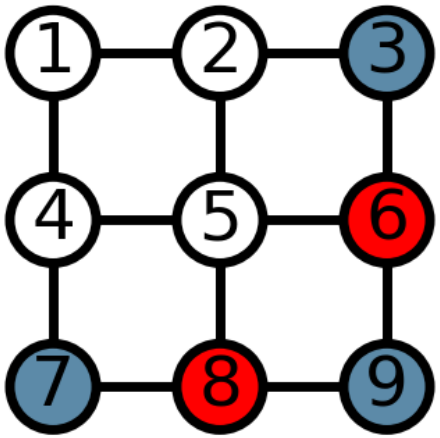


FIG. : Graphe G

$V_2 = (3\ 7\ 9)$
$V_3 = (6\ 8)$

Tri de V_2 par degré dans V_3 :

$deg(3, V_3) = deg(7, V_3) = 1$
$deg(9, V_3) = 2$
donc $V_2 = (3\ 7 \mid 9)$

$\pi' = (1 \mid 3\ 7 \mid 9 \mid 6\ 8 \mid 2\ 4 \mid 5)$

tri π' plus fin

Tri des sommets encore plus fin : propagation du degré

Tri optimal : la propagation de degré ne donne plus de nouveau tri

$$\forall (i,j) \in [\![1,n]\!]^2, \text{ tous les sommets de } V_i \text{ ont le même degré dans } V_j$$
$$\iff \forall (i,j) \in [\![1,n]\!]^2,\ \forall (v,w) \in V_i^2,\ deg(v,V_j) = deg(w,V_j)$$

Soit π un tri, on note :

$$\boxed{R(\pi) \text{ le plus grand}^3 \text{ tri équitable ordonné obtenu à partir de } \pi} \quad (3)$$

[3] défini selon un ordre partiel sur les partitions

Isomorphe cannonique de McKay

$C_M(G)$ l'isomorphe cannonique [4] du graphe G à partir d'un tri π

alors :

$$\boxed{G \cong H \text{ si et seulement si } C_M(G) = C_M(H)} \quad (4)$$

Exemple : pour $\pi = (1 \mid 3\ 7 \mid 9 \mid 6\ 8 \mid 2\ 4 \mid 5)$

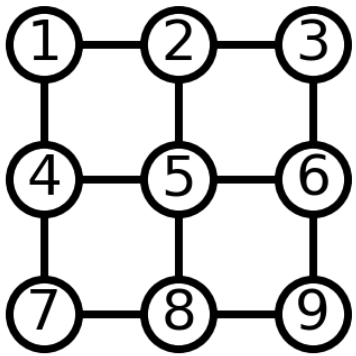


FIG. : Graphe G

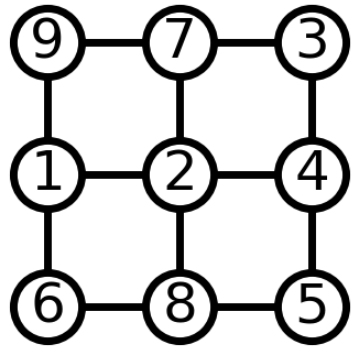


FIG. : Graphe $C_M(G)$

[4] méthode de détermination en annexe

Application sur les protéines

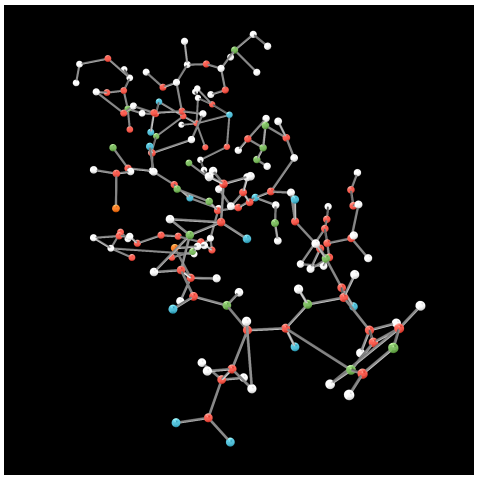


FIG. : Protéine

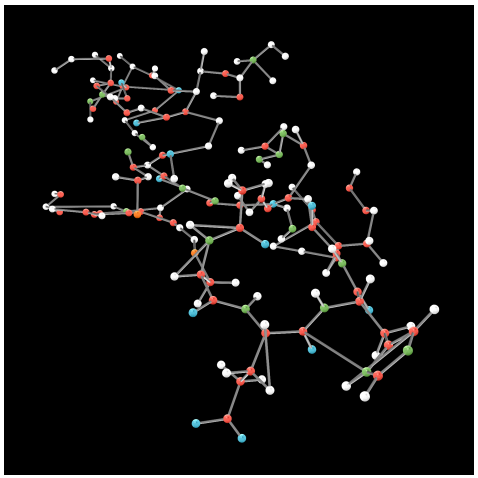


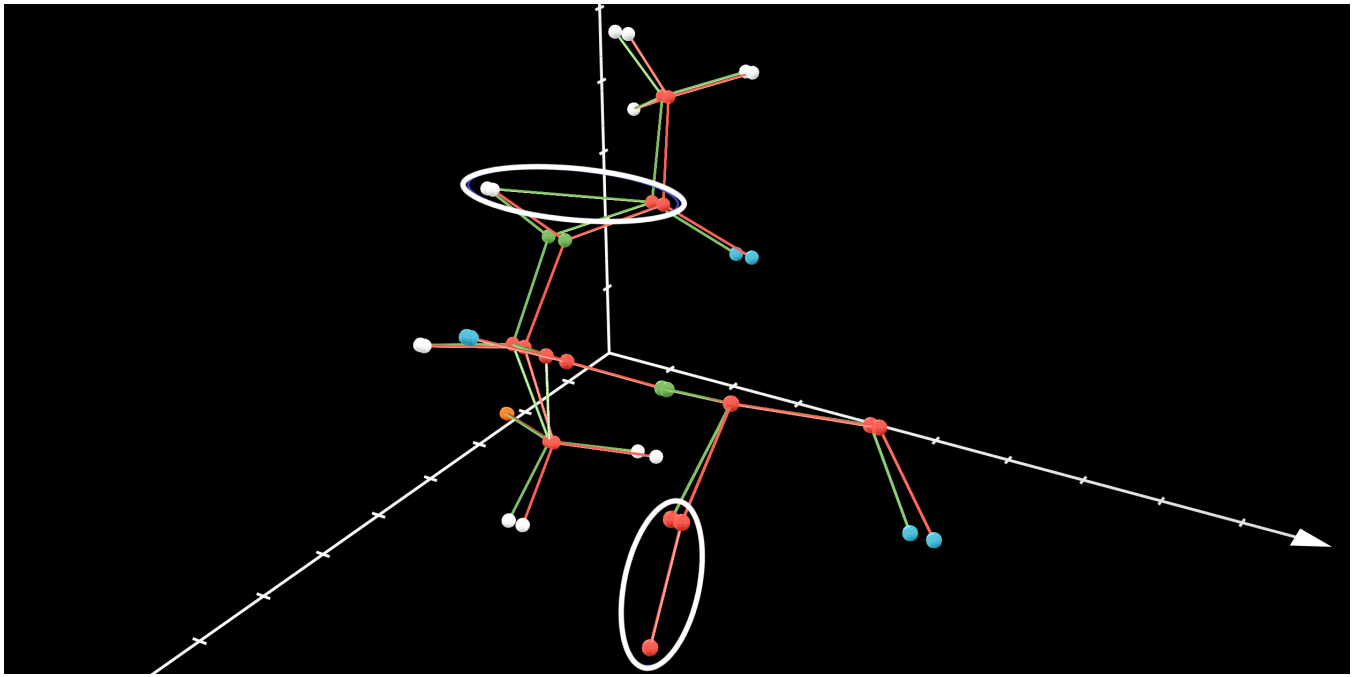
FIG. : Même protéine mais déplacée et numérotation différente des sommets

Temps d'éxecution : pour cette protéine de 171 atomes

tri par degré et atomes	0.002 s
propagation des degrés	7.4 s
test d'isomorphisme	0.3 s
temps total d'éxecution	7.8 s

Résultat : protéines isomorphes et permutation renvoyée correcte
tri + propagation du degré → paquets d'au plus 4 atomes

Recherche du plus grand sous-graphe isomorphe



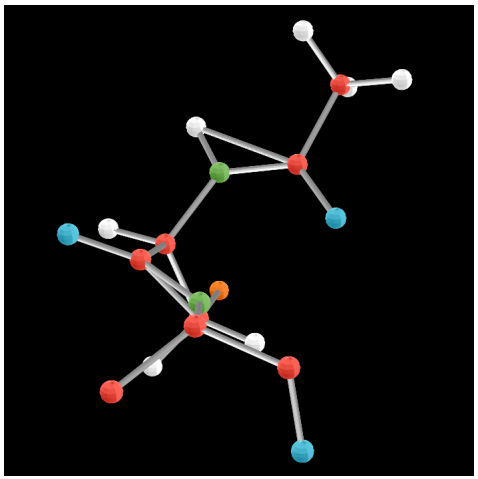


FIG. : Protéine A de 21 atomes (en vert)

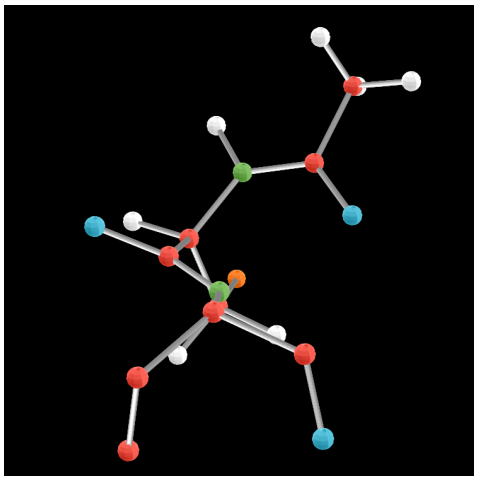


FIG. : Protéine B de 22 atomes (en rouge)

Recherche du plus grand sous-isomorphe

Un sous-graphe isomorphe de taille maximale[5] :

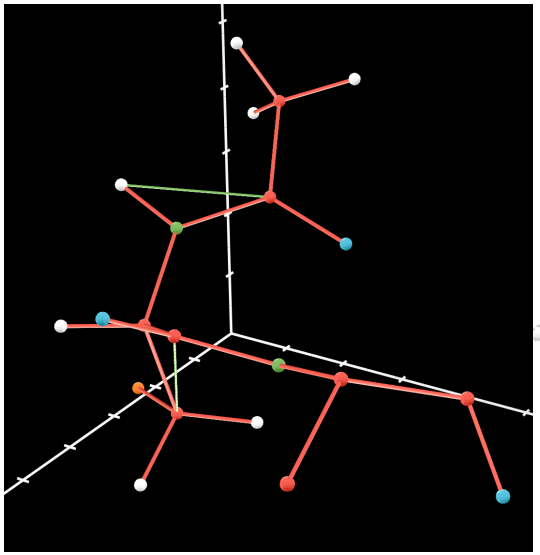


FIG. : Protéine A

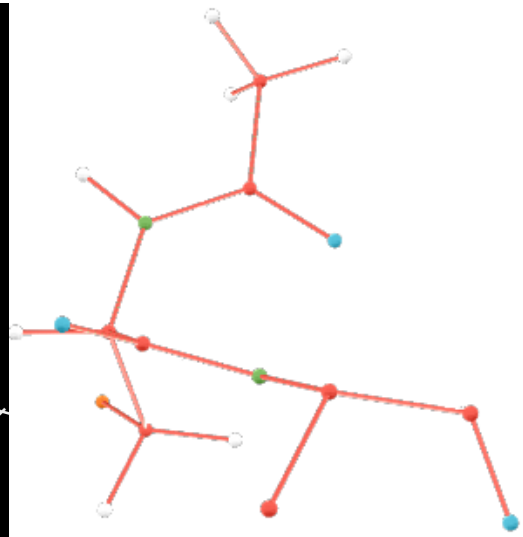


FIG. : Sous-structure commune

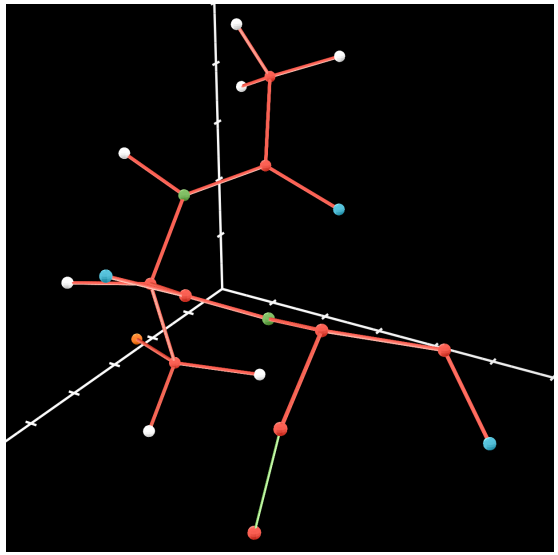


FIG. : Protéine B

→ temps d'éxecution sur ce modèle : 3-4 minutes
→ cas d'égalité, sous-graphes de même taille ?

[5] méthode de recherche en annexe

Position de la sous-structure

- Alignement des sous-structures isomorphes :

- Coefficient de proximité :

branche $\rightarrow$ suite d'arêtes ordonnées
Même coefficient que pour les branches

Comparaison de protéines

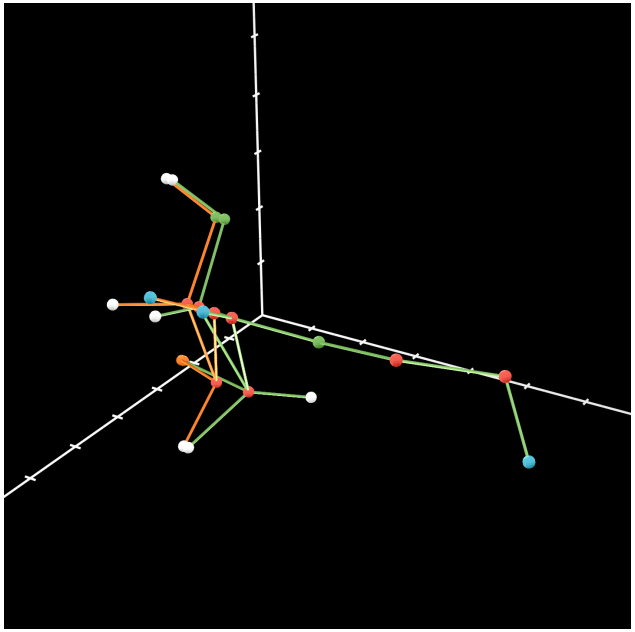
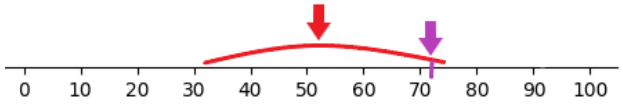


FIG. : Protéines plutôt éloignées



$C_{prox} = 92.9 \quad C_{struct} = 52$

$$\boxed{C = 72}$$

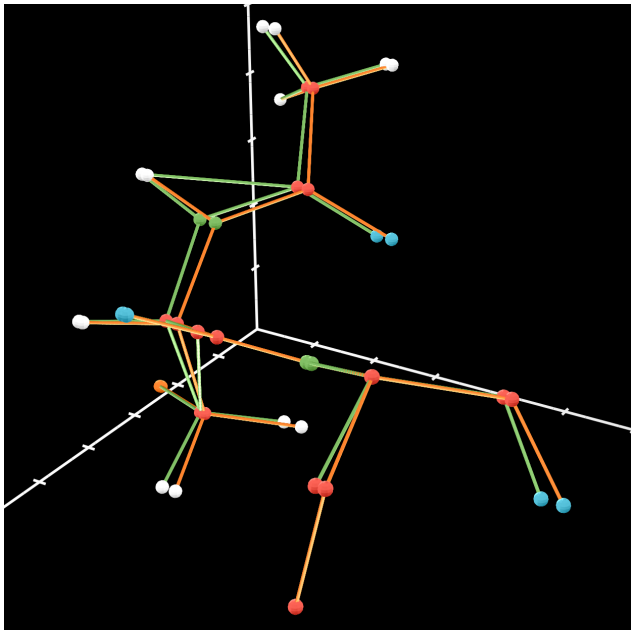
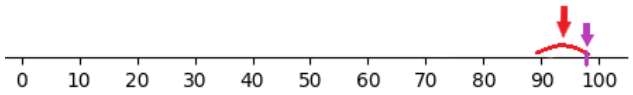


FIG. : Protéines plutôt proches



$C_{prox} = 97.7 \quad C_{struct} = 93$

$$\boxed{C = 96.3}$$

Conclusion

Bilan :

- tri des atomes très classant
- test d'isomorphisme efficace sur les protéines
- méthode recherche du plus grand sous-graphe isomorphe fonctionnelle
- mise en place d'un coefficient de comparaison

Limites :

- lecture partielle du format pdb
- coefficients déterminés de manière empirique
  $\rightarrow$ sur une base d'exemples, ajuster les valeurs
- recherche demande beaucoup de ressources (NP-complet)

Annexe

Arbre de recherche de permutations

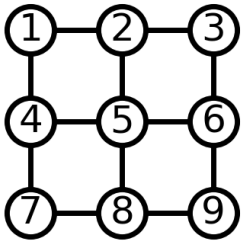


FIG. : Graph G

On considère le tri suivant le degré :
$\pi = (1\ 3\ 7\ 9 \mid 2\ 4\ 6\ 8 \mid 5)$

$R(\pi) = \pi$ est alors la racine de l'arbre

Trouver les fils :

- trouver la première partie V_i d'au moins 2 éléments de π
- pour $v \in V$, on crée **artificiellement** un nouveau tri équitable :
  $\longrightarrow \quad \pi_v = \pi \perp v = R(\ (V_1|..|\ \{v\} \mid V_i \backslash \{v\}\ |..)\ )$
- chacun des tris créés est un fils, on réitère jusqu'à obtenir des tris triviaux (paquets de taille 1)

Arbre de recherche de permutations

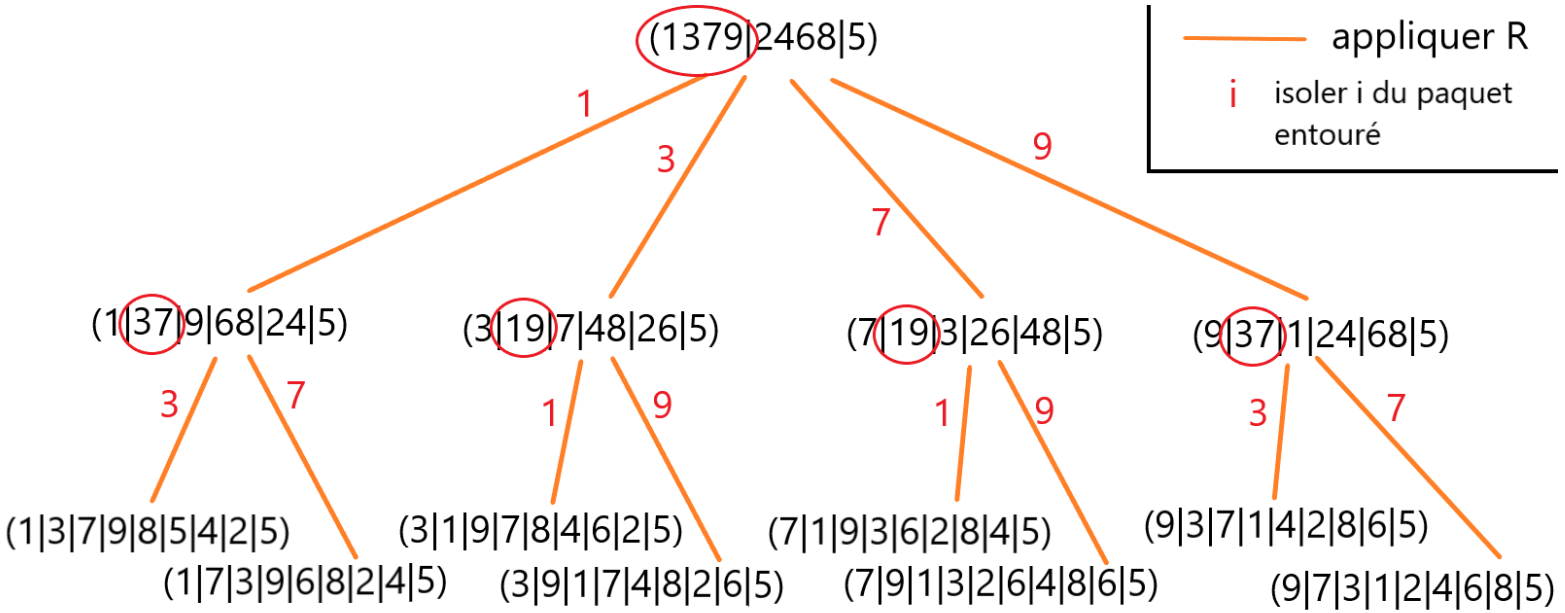


FIG. : Arbre T(G) de racine $\pi = (1\ 3\ 7\ 9 \mid 2\ 4\ 6\ 8 \mid 5)$

Relation d'ordre sur les graphes et isomorphisme cannonique

Ordre total $\preceq$ sur les graphes :

On pose la fonction $i : G \longmapsto i(G)$ telle que :
- $i(G)$ est la séquence binaire $(\mathbb{1}_{(i,j)\in G})$ dans l'ordre lexicographique
- $G \preceq H$ si et seulement si $i(G) \leq i(H)$ en décimal

On pose alors l'**ismorphisme cannonique de McKay** (pour le tri π) :

$$\boxed{C_M(G) = max_{\preceq} \{G^\sigma,\ \sigma \text{ noeud terminal de T(G) de racine } \pi\}} \quad (5)$$

alors :

$$\boxed{G \cong H \text{ si et seulement si } C_M(G) = C_M(H)} \quad (6)$$

Exemple : pour $\pi = (1 \mid 3\ 7 \mid 9 \mid 6\ 8 \mid 2\ 4 \mid 5)$

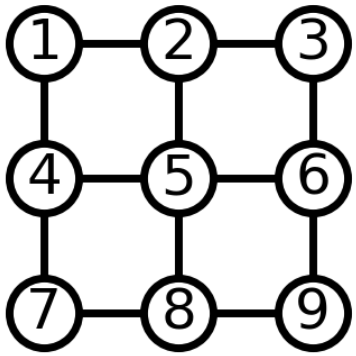


FIG. : Graphe G

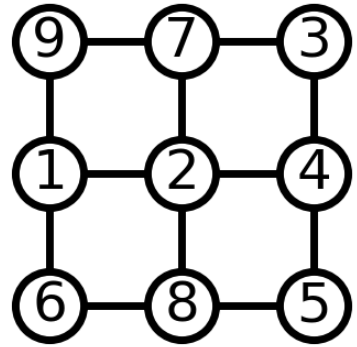


FIG. : Graphe $C_M(G)$

```
from def_protein import *
from generer_protein import *

#lecture pdb

def entier(car):
    car = car.split('.')
    a, b = int(car[0]), int(car[1])
    return round( a + b/10**len(car[1]), 2)

def read(pdb, liaisons=False):
    file = open(pdb)
    li, lp, lm, connect = [], [], [], []
    for line in file:
        if ('ATOM' in line) or ('HETATM' in line):
            l = [x for x in line.split('␣') if x!='' and (x!='\n')
                  and (x!='\\n') and (x!="'\n")]
            if l[0]=='ATOM' or l[0]=='HETATM':
                li.append( int(l[1]) )
                point = [entier(l[6]),entier(l[7]),entier(l[8])]
                lp.append(point)
                lm.append(l[-1])
        if 'CONECT' in line:
            connect.append([int(y) for y in [x for x in line.split
                ('␣') if (x!='') and (x!='CONECT') and (x!='\n')
                and (x!='\\n') and (x!="'\n")]])
    file.close()
    for k in range(len(connect)):#Renumérotation
        for l in range(len(connect[k])):
            connect[k][l] = li.index(connect[k][l])
```

```
    if liaisons :
        connexions = [[] for _ in range(len(li))]
        for c in connect :
            connexions[c[0]] = c[1:]
        P = Protein([Atom(k,lp[k],lm[k]) for k in range(len(li))],
              connexions) #connexions ne fonctionne pas ->
               subtilités du format
        return filtre_RE(P)
    P = Protein([Atom(k,lp[k],lm[k]) for k in range(len(li))], [[]
          for _ in range(len(li))]) #sans connections
    return filtre_RE(P)

def test(pdb) :
    file = open(pdb)
    for line in file :
        print(line)

#écrire

save_folder = "C:\\Users\\Adrien␣Dubois\\Desktop\\TIPE\\2-Code\\
    pdb\\save_prot\\"

def concatstr(liste) :
    s = ''
    for l in liste :
        s += str(l)+'␣'
    return s

def save(prot,nom) :
    file = open(save_folder + nom, 'w')
    file.write('latom\n')
```

```
    for at in prot.latom:
        file.write(str(at.indice) +';'+ concatstr(at.point) +';'+
            str(at.atom)+'\n')
    file.write('connect\n')
    for i in range(prot.nbr):
        file.write(concatstr(prot.connect[i])+'\n')
    file.close()

def recup(nom):
    file = open(save_folder + nom, 'r')
    latom, connect = [], []
    blat, bcon = False, False
    for line in file:
        if 'latom' in line:
            blat = True
        elif 'connect' in line:
            blat, bcon = False, True
        elif blat:
            i, p, a = tuple(line.split(';'))
            i, a = int(i), a[0]
            p = [float(k) for k in (p.split('␣')) if k!='']
            latom.append( Atom(i,p,a) )
        elif bcon:
            l = line.split('␣')[:-1]
            connect.append([int(k) for k in l])
    return Protein(latom, connect)
```

```
from cas_simple_nuage import *

#exemple - nuage point (non lies / lies)

def rda(x=0.5,eps=0.5):
    a, b = x-eps, x+eps
    return (b-a)*rd.random() + a

def generernuage(n=10,xlim=10,ylim=10,zlim=10):
    l = []
    for _ in range(n):
        l.append([xlim*rda(),ylim*rda(),zlim*rda()])
    return nuagepoints(l)

def genererliaisons():#on trie les points selon la diagonale (axe
    n=(1,1,1) passant par O)
    g = generernuage()
    l = []
    for i in range(g.nbr):
        if l==[]:
            l.append([g.liste[i],g.diag[i]])
        else:
            for j in range(len(l)):
                if g.diag[i]<=l[j][1]:
                    l = l[:j] + [g.liste[i],g.diag[i]] + l[j:]
                    break
                if j==n:
                    l.append([g.liste[i],g.diag[i]])
        return nuagepoints([h[0] for h in l])
```

```
def genererliaisonsunif(n, c):
    D = distance([0,0,0],[c, c, c]) #diagonale du cube
    ndiag = [k*(D/(n+1)) for k in range(1,n+1)]
    u = (1/np.sqrt(3)) * np.array( [-1,-1, 1])
    v = (1/np.sqrt(2)) * np.array( [ 1, -1, 0])

    def is_incube(l, c):
        return 0<=l[0]<=c and 0<=l[1]<=c and 0<=l[2]<=c

    def genpointdiag(diag,M):
        while 1:
            x = M * (-1 + 2*rd.random())
            y = M * (-1 + 2*rd.random())
            A = ( diag/np.sqrt(3) * np.array([1,1,1]) )
            B = x * u
            C = y * v
            point = (A+B+C).tolist()
            if is_incube(point, c):
                return point

    def interplancube(diag, c): #donne
        if diag <= D/np.sqrt(3) :
            x = diag * np.sqrt(3)
            return [ [x,0,0], [0,x,0], [0,0,x] ]
        elif diag >= D*(1-np.sqrt(3)) :
            x = c - (D-diag) * np.sqrt(3)
            return [ [x,c,c], [c,x,c], [c,c,x] ]
        return [ [c,0,0] ]
    def distM(diag, c):#definir un distance a la diagonale du
        cube maximale
        P = interplancube(diag, c)
```

```
        A = ( diag/np.sqrt(3) * np.array([1,1,1]) ).tolist()
        return max([distance(A,p) for p in P])

    l = []
    for i in range(n):
        M = distM(ndiag[i], c)
        l.append( genpointdiag(ndiag[i], M) )

    return nuagepoints(l)

#exemple - générer une approximation a partir d'un modele

def decalagex(NA,eps):
    l = []
    for i in range(NA.nbr):
        l.append( [NA.x[i], NA.y[i] + rda(0.1,1)*eps, NA.z[i]] )
    return nuagepoints(l)

#affichage temporaire

N1 = generernuage()
N2 = genererliaisons()
N3 = genererliaisonsunif(10, 10)
N4 = rotation_z(N1)

def show(N, M, relies=False):#points reliés : dans l'ordre de le
    liste
        fig = plt.figure()
        md = Axes3D(fig)
        for a in N.liste:
            md.scatter(a[0],a[1],a[2], linewidths = 4)
```

```
        for a in M.liste :
            md.scatter(a[0],a[1],a[2], linewidths = 4)
        if relies :
            for i in range(N.nbr - 1) :
                md.plot(N.x[i :i+2],N.y[i :i+2],N.z[i :i+2], color='
                    black')
                md.plot(M.x[i :i+2],M.y[i :i+2],M.z[i :i+2], color='
                    blue')
        plt.show()
```

```
from def_protein import *

#generer proteine
#generation 2 : 4 liaisons par carbone, 1 par hygrogène, 3 par
    azote, etc -> pbm de satisfiabilité (incroyable)

def liste_cercle(liste_atom) :
    return [[x.point, 0, x.point.copy(), x.indice] for x in
        liste_atom] #on conserve l'indice initial

def contact(i,j,lc) :
        return distance(lc[i][0] ,lc[j][0]) <= (lc[i][1] + lc[j
            ][1])

def update(lc, n, fixe, trans) : #n=len(lc)
    for i in range(n) :
        count, ref = 0, []
        for j in [k for k in range(n) if k!=i] :
            if contact(i,j,lc) :
```

```
                count += 1
                ref.append(j)
        if count>=2:
            fixe[i] = ref
        if count==1:
            trans[i] = ref[0]

def incrementer(lc, n, fixe, trans, eps, rmax):
    for i in range(n):
        if fixe[i]!=False:
            pass
        elif trans[i]!=False:
            a = lc[trans[i]][0]
            b = lc[i][0]
            AB = np.array(b)-np.array(a)
            lc[i][0] = ( np.array(b) + eps * (AB)/distance(a,b) ).
                tolist()
            lc[i][1] += eps
        else:
            lc[i][1] += eps
    r = max([lc[i][1] for i in range(n)])
    if r > rmax:
        for i in range(n):
            if fixe[i]==False:
                fixe[i] = [trans[i]]

def vertices(lc, fixe, n):
    def maxi(liste):
        if liste==[]:
            return 0
        return max(liste)
```

```
    def rev(couple):
        a,b=couple
        return b,a
    lcouples = [ (tuple(lc[i][2]),tuple(lc[j][2])) for i in range(
        n) for j in fixe[i] ]
    n, k = len(lcouples), 0
    while k < n:
        if (lcouples[k] in lcouples[k+1:]) or (rev(lcouples[k]) in
            lcouples[k+1:]):
            lcouples.pop(k)
            n += -1
        else:
            k += 1
    return lcouples

def tri_denombrement(l,N): #N nbr sommets
    compt = [False]*N
    for x in l:
        compt[x] = True
    return [x for x in range(N) if compt[x]]


#si le graphe n'est pas connexe, relier les composantes connexes

def connexe(g): #retounre partition des sommets
    n = len(g)
    T = [False for _ in range(n)]
    a_visiter = [0]
    T[0] = True
    comp_connexes = []
    for i in range(n):
```

```
        if not T[i]:
            a_visiter = [i]
            T[i] = True
            comp = []
            while a_visiter != []:
                s = a_visiter.pop(0)
                comp.append(s)
                for x in g[s]:
                 if not T[x]:
                    T[x] = True
                    a_visiter.append(x)
            comp_connexes.append(comp)
    return comp_connexes

def dmin(l1, l):
    dm, xm, ym = distance(lc[l1[0]][2],lc[l[0][0]][2]), l1[0], l
        [0][0]
    for l2 in l:
        for x in l1:
            for y in l2:
                d = distance(lc[x][2],lc[y][2])
                if d<dm:
                    dm, xm, ym = d, x, y
    return xm, ym, dm

def dmax(l,lc):
    n = len(l)
    dm, xm, ym = distance(lc[l[0]][2],lc[l[1]][2]), l[0], l[1]
    for i in range(n-1):
        for j in range(i+1,n):
            x, y = l[i], l[j]
```

```
                d = distance(lc[x][2],lc[y][2])
                if d>dm:
                    dm, xm, ym = d, x, y
    return xm, ym, dm

def relier(g, lc):#sortie -> un graphe connexe
    c = connexe(g)
    n = len(c)
    while n>1:
        x, y, d = dmin(c[0],c[1:])
        g[ x].append( y )
        g[ y ].append( x )
        c = connexe(g)
        n += -1

#retirer cycles

#A) Trouver cycles -> liste des cycles représentés par une liste
     des indices des arêtes

def cycle_min(s0,g,n):
    chemins = [ [s0] ]
    new_chemins = []
    while chemins!=[]:
        new_chemins = []
        for chemin in chemins:
            for s in g[ chemin[-1] ]:
                if not s in chemin:
                    if s0 in g[s] and len(chemin)>1:
                        chemin.append(s)
                        return tri_denombrement(chemin,n)
```

```
                    new_chemins.append(chemin+[s])
        chemins = new_chemins
    return []

def sans_repet(l):
    n = len(l)
    if n<=1:
        return l
    if l[0] in l[1:] or l[0]==[]:
        return sans_repet(l[1:])
    return [l[0]]+sans_repet(l[1:])

def cycles(g,n):
    lcyclesmin = []
    for s in range(n):
        c = cycle_min(s,g,n)
        if c!=[]:
            lcyclesmin.append( c )
    return sans_repet(lcyclesmin)

#B) Pour chaque cycle, enlever l'arete de distance maximale ->
    retourne un graphe connexe dit arbre (connexe sans cycle)

# def indice_max(l):
#     n = len(l)
#     if n==0:
#         return -1
#     mini, i0 = l[0], 0
#     for i in range(n):
#         if l[i] > mini:
#             mini, i0 = l[i], i
```

```
#     return i0

def indmaxparmi(l, lt, n):
    m = max([l[i] for i in range(n) if not lt[i]])
    for i in range(n):
        if l[i]==m and not lt[i]:
            return i

def tri_ind(l):
    n = len(l)
    lt = [False for _ in range(n)]
    lp = []
    while False in lt:
        im = indmaxparmi(l, lt, n)
        lt[im] = True
        lp.append(im)
    return lp


def enlever_poids_max(cycle, aretes_enlevees, lpoints, g):
    c = len(cycle)
    l_aretes = [(cycle[i],cycle[j]) for i in range(c-1) for j in
        range(i+1,c) if (cycle[j] in g[cycle[i]]) and not ((cycle
        [i],cycle[j]) in aretes_enlevees) and not ((cycle[j],
        cycle[i]) in aretes_enlevees)]
    ldist = [distance(lpoints[i][2],lpoints[j][2]) for (i,j) in
        l_aretes]
    im = indice_max(ldist)
    if im!=-1:
        ti = tri_ind(ldist)
        n = len(l_aretes)
```

```
        i = 0
        while i<n and not est_connexe_sans(g, l_aretes[ti[i]]):
            i+=1
        if not est_connexe_sans(g, l_aretes[ti[n-1]]):
            return None
        a, b = l_aretes[ti[i]]
        aretes_enlevees.append((a,b))
        aretes_enlevees.append((b,a))

def enlever_cycles(g, lpoints, n):
    print('démarrage')
    C = cycles(g,n)
    print('fin')
    print('␣')
    aretes_enlevees = []
    for cycle in C:
        enlever_poids_max(cycle, aretes_enlevees, lpoints, g)
    for (a,b) in aretes_enlevees:
        if b in g[a]:
            g[a].remove(b)
        if a in g[b]:
            g[b].remove(a)
    return aretes_enlevees

#génération

def gen_carbones(prot):
    '''Entrée : protéine sans liaisons
       Sortie : protéine avec liaisons carbones'''

    carbones = Protein(prot.extraire_molecule('C'), [])
```

```
    lc = liste_cercle(carbones)
    n = len(lc) #nbr de carbones
    eps = ecart_type([distance(lc[i][0],lc[j][0]) for i in range(n
        ) for j in range(i+1,n)])/100
    fixe = [False]*n # future liste d'adjacence du graphe carbone
    trans = [False]*n
    rmax = dmax([i for i in range(prot.nbr)],lc)

    while False in fixe:
        incrementer(lc, n, fixe, trans, eps, rmax)
        update(lc, n, fixe, trans)

    #ici, les carbones sont reliés, mais composantes non connexes
        et/ou cycles qui correspondent à des arêtes inutiles

    relier(fixe, lc)
    enlever_cycles(fixe, lc, n)

    #on a alors obtenu un "arbre couvrant" reliant les carbones,
        reste à convertir en format Protein
    #on reporte les connections à la protéine entière
    connections = [[] for _ in range(prot.nbr)]
    for i in range(n):
        for j in fixe[i]:
            connections[ lc[i][3] ].append( lc[j][3] )

    return Protein(prot.latom, connections)

#ALTERNATIVE (plus simple) : on traite de la même manière toutes
    les molécules
```

```
def gen_uniforme(prot):
    lc = liste_cercle(prot.latom)
    eps = ecart_type([distance(lc[i][0],lc[j][0]) for i in range(
        prot.nbr-1) for j in range(i+1,prot.nbr)])/100
    fixe = [False]*prot.nbr # future liste d'adjacence du graphe
    trans = [False]*prot.nbr
    rmax = dmax([i for i in range(prot.nbr)],lc)[2]

    while False in fixe:
        incrementer(lc, prot.nbr, fixe, trans, eps, rmax)
        update(lc, prot.nbr, fixe, trans)
    print('ok')
    relier(fixe, lc)
    print('ok1')
    enlever_cycles(fixe, lc, prot.nbr)
    print('ok2')

    return Protein(prot.latom, fixe)

#supprimer les molécules seules

import sys
sys.path.insert(0, "anim2D")
from gencarbones2D import connexe

def filtre(prot,li):
    latom = [Atom(li.index(k), prot.latom[k].point, prot.latom[k].
        atom) for k in li]
    connect = [[li.index(v) for v in inter2(li,prot.connect[s])]
        for s in li]
    return Protein(latom, connect)
```

```
def filtre_liaisons(prot):
    comp = connexe(prot.connect)
    comp_max = comp[indice_max([len(c) for c in comp])]
    li = [] #liste des sommets gardés
    for s in comp_max:
        if not s in li:
            li.append(s)
    return filtre(prot,li)

def filtre_RE(prot):
    li = [k for k in range(prot.nbr) if prot.latom[k].atom in ['C'
        ,'N','O','H','S']]
    return filtre(prot,li)

def filtre_nbr(prot,nbr_lim):
    li = [i for i in range(min(nbr_lim,prot.nbr))]
    return filtre(prot,li)


#plier la proteines -> diapo 2 structures isom positions diff

def indice_point_mileu(prot):
    paires = [(i,j) for i in range(prot.nbr) for j in range(prot.
        nbr)]
    l_diam = [distance(prot.liste[i], prot.liste[j]) for (i,j) in
        paires]
    i,j = paires[indice_max(l_diam)] #indices des points les plus
        éloignés
    pos_milieu = [ (prot.liste[i][k]+prot.liste[j][k])/2 for k in
        [0,1,2] ]
```

```
    return (prot.latom[ indice_min([distance(prot.liste[i],
        pos_milieu) for i in range(prot.nbr)]) ]).indice

def plier(prot):
    im = indice_point_mileu(prot)
    select = [i for i in range(prot.nbr) if prot.liste[i][2]<=prot
        .liste[im][2]] #points au dessus de i
    #lrot = rotation_z([prot.liste[i] for i in select],prot.liste[
        im][0],prot.liste[im][1]) pas de changements ?
    lpos, k = [], 0
    for i in range(prot.nbr):
        if i in select:
            lpos.append([prot.liste[i][0],prot.liste[i][1]+(prot.
                liste[i][2]-prot.liste[im][2]), prot.liste[i
                ][2]])
            k+=1
        else:
            lpos.append(prot.liste[i])
    return Protein( [Atom((prot.latom[i]).indice, lpos[i], (prot.
        latom[i]).atom) for i in range(prot.nbr)], prot.connect )

#liason proche aléatoire (minimiser la taille des paquets)

def liaison_proche(prot,adj,i):
    im = indice_min([distance(prot.latom[i].point, prot.latom[j].
        point) for j in range(prot.nbr) if (j!=i and not j in adj
        [i])])
    adj[i].append(im)
    adj[im].append(i)

def ajouter_liaisons(prot,adj,p=0.1): #p proportion
```

```
    p = int(np.ceil(p * prot.nbr))
    for _ in range(p):
        i = rd.randint(0,prot.nbr-1)
        liaison_proche(prot,adj,i)

#méthode 2 : arbre couvrant

from arbre_couvrant import *

def gen_couvrant(prot, ajout=0):#ajout entre 0 et 1 proportion de
    liaisons en plus
    mat_dist = prot.matrice_dist()
    g = [[(j,mat_dist[i][j]) for j in range(prot.nbr)] for i in
        range(prot.nbr)]
    liste_couvrante = ACM(g)
    adj = [[] for _ in range(prot.nbr)]
    for (i,j) in liste_couvrante:
        adj[i].append(j)
        adj[j].append(i)

    if ajout!=0:
        ajouter_liaisons(prot,adj,ajout)

    return Protein(prot.latom, adj)


#générer une permuation de la protéine -> inverser k to n-1-k

def permut_prot(prot):
    def inv(k):
        return prot.nbr-1-k
```

```
    latom_inv = [Atom(inv(prot.latom[inv(i)].indice), prot.latom[
        inv(i)].point, prot.latom[inv(i)].atom) for i in range(
        prot.nbr)]
    connect_inv = [[inv(s) for s in prot.connect[inv(i)]] for i in
         range(prot.nbr)]
    return Protein(latom_inv, connect_inv)

#légèrement bouger les positions

def shuffle_trans(prot):
    for i in range(prot.nbr):
        prot.latom[i].point[1] += 1 * rd.random()
```

```
from heapq import *
import numpy as np
import matplotlib.pyplot as plt

# G = [[(4 , 60) , (5, 100)], [(2, 20) , (3, 30) , (4 ,40)],
# [(1 , 20) , (3, 10)], [(1, 30) , (2, 10) , (4, 50)],
# [(0 , 60) , (1, 40) , (3, 50) , (5, 70)], [(0, 100) , (4, 70) ]]

def taille(g):
    return len(g)

def voisins(g, s):
    return g[s]

def aretes_poids(g, s, vu):
    return [(p,s,i) for (i,p) in g[s] if (vu[i]+vu[s])==1]
```

```
def ss_doublons(lcouples):
    def rev(couple):
        p,a,b=couple
        return p,b,a
    n, k = len(lcouples), 0
    while k < n:
        if (lcouples[k] in lcouples[k+1:]) or (rev(lcouples[k]) in
            lcouples[k+1:]):
            lcouples.pop(k)
            n += -1
        else:
            k += 1
    return lcouples


def ajout(file, trip):
    heappush(file, trip)

def ACM(g):
    n = len(g)
    vu = [True]+[False]*(n-1)
    poids_min = []
    while len(poids_min)<(n-1):
        aretes = []
        for i in range(n):
            if not vu[i]:
                ap = aretes_poids(g,i,vu)
                for trip in ap:
                    ajout(aretes, trip)
        p,i,j = heappop(aretes)
```

```
            vu[i], vu[j] = True, True
            poids_min.append( (i,j) )
    return poids_min
```

```
import networkx as nx
import matplotlib.pyplot as plt

G1 = nx.Graph()
S1 = [(1,4),(1,6),(1,2),(2,5),(2,3),(3,6),(3,4),(4,5),(5,6)]
for (i,j) in S1:
    G1.add_edge(i, j)

G2 = nx.Graph()
S2 = [(1,2),(1,4),(1,6),(2,3),(2,5),(3,4),(3,6),(4,5),(5,6)]
for (i,j) in S2:
    G2.add_edge(i, j)

sigma = [0, 3, 6, 5, 4, 1, 2] #0 pour l'indexation
S_permut = [(sigma[i],sigma[j]) for (i,j) in S1]
G_permut = nx.Graph()
for (i,j) in S_permut:
    G_permut.add_edge(i, j)

# explicitly set positions
pos1 = {1: (0, 0), 2: (1,0), 3: (2,1), 4: (1,2), 5: (0,2), 6:
     (-1,1)}
pos2 = {1: (0,2), 2: (0.9,1.4), 3: (1.5,0.5), 4: (2,2), 5: (2.5,1)
     , 6: (1,-1)}
pos_permut = {sigma[k]: pos1[k] for k in pos1.keys()}
```

```
options = {
    "font_size": 36,
    "node_size": 2000,
    "node_color": "white",
    "edgecolors": "black",
    "linewidths": 5,
    "width": 5,
}

#nx.draw_networkx(G1, pos1, **options)
nx.draw_networkx(G_permut, pos_permut, **options)
#nx.draw_networkx(G2, pos2, **options)

# Set margins for the axes so that nodes aren't clipped
ax = plt.gca()
ax.margins(0.20)
plt.axis("off")
plt.show()
```

```
pos1 = {1: (0, 0), 2: (1,0), 3: (2,1), 4: (1,2), 5: (0,2), 6:
    (-1,1)}
pos2 = {1: (0,2), 2: (0.9,1.4), 3: (1.5,0.5), 4: (2,2), 5: (2.5,1)
    , 6: (1,-1)}

def adj(lar):
    m = max([max(c) for c in lar])
    adja = [[] for _ in range(m)]
    for (i,j) in lar:
        adja[i-1].append(j-1)
        adja[j-1].append(i-1)
    return m, adja

n1, S1 = adj([(1,4),(1,6),(1,2),(2,5),(2,3),(3,6),(3,4),(4,5)
    ,(5,6)])
n2, S2 = adj([(1,2),(1,4),(1,6),(2,3),(2,5),(3,4),(3,6),(4,5)
    ,(5,6)])

tat1 = ['C','H','C', 'C', 'H','C']
tat2 = ['O','C', 'H', 'H', 'N','S']
lat1 = [Atom(i,(*pos1[i+1],np.random.random()),tat1[i]) for i in
    range(n1)]
lat2 = [Atom(i,(*pos2[i+1],np.random.random()),tat1[i]) for i in
    range(n1)]
lat3 = [Atom(i,(*pos2[i+1],np.random.random()),tat2[i]) for i in
    range(n1)]
prot_isom1 = Protein(lat1,S1)
prot_isom2 = Protein(lat2,S2)
prot_isom3 = Protein(lat3,S2)
```

```
nr, Sr = adj([(1,4),(2,4),(4,5),(3,5)])
posr = {1: (0, 0), 2: (2,0), 3: (4,0), 4: (1,2), 5: (3,2)}
tatr = ['O','O','C','O','C']
latr = [Atom(i,(*posr[i+1],np.random.random()),tatr[i]) for i in
    range(nr)]
prot_tri1 = Protein(latr,Sr)
nt, St = adj([(1,4),(2,4),(4,5),(3,5)])
post = {1: (-2, 6), 2: (2,6), 3: (0,0), 4: (0,4), 5: (0,2)}
latt = [Atom(i,(*post[i+1],np.random.random()),tatr[i]) for i in
    range(nr)]
prot_tri2 = Protein(latt,St)

#test pour isomorphisme

pdb_ex = "C:\\Users\\Adrien␣Dubois\\Desktop\\TIPE\\2-Code\\pdb\\
    prot1.pdb␣"

def gen_isom_prot(liaisons_supp=0.1,replace=False):
    new_prot_ex_isom1 = gen_couvrant(read(pdb_ex),liaisons_supp)
    new_prot_ex_isom2 = plier(permut_prot(new_prot_ex_isom1))
    if replace:
        save(new_prot_ex_isom1,'prot_ex_isom1')
        save(new_prot_ex_isom2,'prot_ex_isom2')
    else:
        save(new_prot_ex_isom1,'prot_ex_isom1prime')
        save(new_prot_ex_isom2,'prot_ex_isom2prime')

#exemples proteines

prot_ex_isom1 = recup('prot_ex_isom1')
prot_ex_isom2 = recup('prot_ex_isom2')
```

```
#base de protéines

from os import listdir
from generer_protein import *

path = 'C:\\Users\\Adrien Dubois\\Desktop\\TIPE\\2-Code\\pdb'
l_pdb = [path+'\\'+x for x in listdir(path) if '.pdb' in x]
l_prot = [gen_couvrant(read(pdb),0.1) for pdb in l_pdb]
```

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from geometrie_et_aux import *
from mesure_Rcoeffs import *

#class

class nuagepoints:

    def __init__(self,l):
        self.nbr = len(l)
        self.liste = l
        self.x = [i[0] for i in l]
        self.y = [i[1] for i in l]
        self.z = [i[2] for i in l]
        self.bords = ( [min(self.x),min(self.y),min(self.z)], [max
            (self.x),max(self.y),max(self.z)] ) #coordonnées d'un
             pavé englobant les points
        self.dist = [distance(self.liste[i],self.liste[i+1]) for i
             in range(self.nbr-1)]
        self.len = sum(self.dist)
        self.diag = [np.dot(self.liste[i],[1,1,1])/np.sqrt(3) for
            i in range(self.nbr)]

    def write(self):
        for x in self:
            print(x[0], x[1], x[2])

    def show(self, relies=False):#points reliés : dans l'ordre de
         le liste
```

```
        fig = plt.figure()
        md = Axes3D(fig)
        for a in self.liste:
            md.scatter(a[0],a[1],a[2], linewidths = 4)
        if relies:
            for i in range(self.nbr - 1):
                md.plot(self.x[i:i+2],self.y[i:i+2],self.z[i:i+2],
                        color='black')
        plt.show()


#exemple - nuage point (non lies / lies)

def rda(x=0.5,eps=0.5):
    a, b = x-eps, x+eps
    return (b-a)*rd.random() + a

def generernuage(n=10,xlim=10,ylim=10,zlim=10):
    l = []
    for _ in range(n):
        l.append([xlim*rda(),ylim*rda(),zlim*rda()])
    return nuagepoints(l)

def genererliaisons():#on trie les points selon la diagonale (axe
    n=(1,1,1) passant par O)
    g = generernuage()
    l = []
    for i in range(g.nbr):
        if l==[]:
            l.append([g.liste[i],g.diag[i]])
        else:
```

```
            for j in range(len(l)):
                if g.diag[i]<=l[j][1]:
                    l = l[:j] + [g.liste[i],g.diag[i]] + l[j:]
                    break
                if j==n:
                    l.append([g.liste[i],g.diag[i]])
        return nuagepoints([h[0] for h in l])

def genererliaisonsunif(n, c):
    D = distance([0,0,0],[c, c, c]) #diagonale du cube
    ndiag = [k*(D/(n+1)) for k in range(1,n+1)]
    u = (1/np.sqrt(3)) * np.array( [-1,-1, 1])
    v = (1/np.sqrt(2)) * np.array( [ 1, -1, 0])

    def is_incube(l, c):
        return 0<=l[0]<=c and 0<=l[1]<=c and 0<=l[2]<=c

    def genpointdiag(diag,M):
        while 1:
            x = M * (-1 + 2*rd.random())
            y = M * (-1 + 2*rd.random())
            A = ( diag/np.sqrt(3) * np.array([1,1,1]) )
            B = x * u
            C = y * v
            point = (A+B+C).tolist()
            if is_incube(point, c):
                return point

    def interplancube(diag, c): #donne
        if diag <= D/np.sqrt(3) :
            x = diag * np.sqrt(3)
```

```
            return [ [x,0,0], [0,x,0], [0,0,x] ]
        elif diag >= D*(1-np.sqrt(3)) :
            x = c - (D-diag) * np.sqrt(3)
            return [ [x,c,c], [c,x,c], [c,c,x] ]
        return [ [c,0,0] ]
    def distM(diag, c):#definir un distance a la diagonale du
        cube maximale
        P = interplancube(diag, c)
        A = ( diag/np.sqrt(3) * np.array([1,1,1]) ).tolist()
        return max([distance(A,p) for p in P])

    l = []
    for i in range(n):
        M = distM(ndiag[i], c)
        l.append( genpointdiag(ndiag[i], M) )

    return nuagepoints(l)

def rot_z(N,x=5,y=5,theta=-np.pi/2):
    liste = N.liste
    lr = rotation_z(N.liste,x,y,theta)
    return nuagepoints(lr)


#exemple - générer une approximation a partir d'un modele

def decalagex(NA,eps,offset=0):
    l = []
    for i in range(NA.nbr):
        l.append( [NA.x[i], NA.y[i] + rda(0.1,1)*eps + offset, NA.
            z[i]] )
```

```
    return nuagepoints(l)

#affichage temporaire

N1 = generernuage()
N2 = genererliaisons()
N3 = genererliaisonsunif(10, 10)
N5 = genererliaisonsunif(10, 10)
N6 = decalagex(N3,1)
#N4 = rot_z(N1)
```

```
from geometrie_et_aux import *

class Graph :

    def __init__(self, vertices, connect):#connexions liste de
        taille len(lmol) où connexions[i] est une liste des
        indices des molecules connectées
        self. vertices = vertices #majoritairement [|0,n-1|]
        self.nbr = len(vertices)
        self.connect = [sans_repet(l) for l in connect] #liste d'
            adj

    def edges(self):#liste des aretes : non-orienté
        def maxi(liste):
            if liste==[]:
                return 0
            return max(liste)
        def rev(couple):
```

```
            a,b=couple
            return b,a
        lcouples = [ (self.vertices[i],self.vertices[j]) for i in
             range(self.nbr) for j in self.connect[i] ]
        n, k = len(lcouples), 0
        while k < n:
            if (lcouples[k] in lcouples[k+1:]) or (rev(lcouples[k
                ]) in lcouples[k+1:]):
                lcouples.pop(k)
                n += -1
            else:
                k += 1
        return lcouples

    def ldegre(self):
        return [len(self.connect[i]) for i in range(self.nbr)]

    def sort_by_degre(self):#liste tq l[i] ensemble des sommets
         de degre i
        ldeg = self.ldegre()
        ld = [ [s for s in range(self.nbr) if ldeg[s]==i] for i in
             range(max(ldeg)+1) ]
        return [l for l in ld if l!=[]]

G = Graph([0,1,2],[[1],[2],[0]])
H = Graph([0,1,2],[[2,1],[0],[1]])
```

```
from geometrie_et_aux import *
from graphisomorphism import *
from time import time

class Atom :

    def __init__(self, i, position, atom):#par defaut i est l'
        indice dans la liste latom
        self.indice = i
        self.x = position[0]
        self.y = position[1]
        self.z = position[2]
        self.point = position
        self.atom = atom

class Protein :

    def __init__(self, latom, connections):#connexions liste de
        taille len(latom) où connexions[i] est une liste des
        indices des molecules connectées
        self.latom = latom
        self.liste = [atom.point for atom in self.latom]
        self.nbr = len(latom)
        self.connect = [sans_repet_tri(c) for c in connections]

    def voisins(self, i):
        return self.connect[i]

    def ldegre(self):
        return [len(self.voisins(i)) for i in range(self.nbr)]
```

```
    def sort_by_degre(self):#liste tq l[i] ensemble des sommets
         de degre i
        ldeg = self.ldegre()
        return [ [s for s in range(self.nbr) if ldeg[s]==i] for i
             in range(max(ldeg)+1) ]

    def matrice_dist(self):
        mat = [[0]*self.nbr for _ in range(self.nbr)]
        for i in range(self.nbr):
            for j in range(i+1,self.nbr):
                mat[i][j] = distance(self.latom[i].point, self.
                     latom[j].point)
                mat[j][i] = mat[i][j]
        return mat

    def extraire_molecule(self, at):
        return [self.latom[i] for i in range(self.nbr) if self.
             latom[i].atom == at]

    def enum_molecule(self, atom):#renvoie un dictionnaire
        num_atom = {}
        for m in self.latom:
            if not m.atom in s:
                num_atom[ m.atom ] = 0
            num_atom[ m.atom ] += 1
        return num_atom

    def liaisons_sans_doublons(self):
        def maxi(liste):
            if liste==[]:
```

```
                return 0
            return max(liste)
        def rev(couple):
            a,b=couple
            return b,a
        lcouples = [ (self.latom[i],self.latom[j]) for i in range(
            self.nbr) for j in self.connect[i] ]
        n, k = len(lcouples), 0
        while k < n:
            if (lcouples[k] in lcouples[k+1:]) or (rev(lcouples[k
                ]) in lcouples[k+1:]):
                lcouples.pop(k)
                n += -1
            else:
                k += 1
        return [(m1.point,m2.point) for (m1,m2) in lcouples]

    def liaisons_sans_doublons_indice(self):
        def maxi(liste):
            if liste==[]:
                return 0
            return max(liste)
        def rev(couple):
            a,b=couple
            return b,a
        lcouples = [ (i,j) for i in range(self.nbr) for j in self.
            connect[i] ]
        n, k = len(lcouples), 0
        while k < n:
            if (lcouples[k] in lcouples[k+1:]) or (rev(lcouples[k
                ]) in lcouples[k+1:]):
```

```
                lcouples.pop(k)
                n += -1
            else :
                k += 1
        return lcouples
```

```
#naif -> fact(|S|)

def test_isomorphism(G, H, f):
    for i in range(G.nbr):
        if not [f[j] for j in G.connect[ f[i] ]] == H.connect[i]:
            break
        if i==(G.nbr-1):
            return True, f
    return False, []

def test_isomorphism2(G,H,f,g):
    for i in range(G.nbr):
        if not [f[j] for j in G.connect[ f[i] ]] == [g[j] for j in
              H.connect[ g[i] ]]:
            break
        if i==(G.nbr-1):
            return True, composee(inverse(g),f)
    return False, []

def isomorphism(G,H):
    if G.nbr != H.nbr:
        return False, []
    lmf = permutations(G.nbr)
    for f in lmf:
        res = test_isomorphism(G, H, f)
        if res != (False, []):
            return res
    return False, []
```

```
#v2 -> choix d'invariants (degré, type du sommet,  pour
     permutations en paquet)

#tris des sommets

def inter2(l1,l2):
    return [i for i in l1 if i in l2]

def intersection(l, n):#l une liste de liste d'indices, n = len(l
    )
    i0 = indice_min([len(x) for x in l])
    inter, i = l[i0], 0
    while inter != [] and i<n:
        inter = inter2(inter,l[i])
        i += 1
    return inter

def cas_vide(Li,numi):
    if Li==[]:
        return []
    return Li[numi]

def assoc_canonique(L):#L est une liste de partition de [0,n-1],
     attention ordre important / on combine len(L) tris de sommets
      pour créer un tri encore plus efficace de manière unique
    tri = [] #nouveau tri qui combine tous les autres
    l_len, tl = [max(0,len(l)-1) for l in L], len(L)
    num, i =  [-1]+[0]*(tl-1), 0
    while i<tl:
        if num[i]<l_len[i]:
            num[i] += 1
```

```
            liste_inter = [ cas_vide(L[i],num[i]) for i in range(
                tl)]
            x = intersection(liste_inter, tl)
            if x != []:
                tri.append(x)
            i = 0
        while i<tl and num[i]==l_len[i]:
            num[i] = 0
            i += 1
    return tri

#A] on fixe la taille max d'un paquet et on calcule p : toutes les
     permutations de [1,n] pour n allant de 0 à taille max (0 <-
    [], 1 <- [[0]], ...)
#B] pour un paquet de taille k, k! permutaions possibles -> elles
    sont numérotées dans p[k] liste de ces k! permuatation
#   Ainsi, si un numéro est une suite d'indice (représentée par
    une liste) des permutaions d'un paquet, on itère sur tous les
     numéros
#C] Pour chaque numéro, on crée alors une bijection des sommets de
     G vers H, qui conserve les invariants des sommets


def concatenation(num, triG, triH, nb_paquets, tailles_paquets, p
    ): #permut dynamique p variable globale(non), num une
    numérotation, lentri la longueur(commune) des listes,
    l_lentri liste des tailles des paquets
    maxi_nbr = max([max(e) for e in triG])+1
    concat = [0]*maxi_nbr
    for i in range(nb_paquets): #parcours des paquets de tri
```

```
        for j in range(tailles_paquets[i]):#parcours des élément
            d'un même paquet et association avec la permutation
            choisie par num
            j_permut = p[tailles_paquets[i]][num[i]][j]
            concat[ triG[i][j] ] = triH[i][j_permut] # sommet j
                associé au j-ieme élément de la permutation num[i
                ] du paquet i
    return concat #liste de G.nbr éléments qui a un sommet i de G
        associe un sommet concat[i] de H

def recherche_isom(G, H, triG, triH):#tri une partition de [1,n]
    tailles_paquets = [len(ti) for ti in triG] #liste de la taille
        des paquets
    tri_imax = [fact(n)-1 for n in tailles_paquets] #liste du
        nombre de permutations par paquet #/!\ fact(n) - 1
    nmax = max(tailles_paquets) #taille du plus gros paquet
    p = permut_dynamique(nmax) #tableau fixe d'élément k ayant la
        liste des permutations de [1,k] (k e [0,nmax])
    t = len(triG)
    num, i = [-1]+[0]*(t-1), 0
    while i<t:
        if num[i]<tri_imax[i]:
            num[i] += 1
            #traitement
            x = test_isomorphism(G, H, concatenation(num, triG,
                triH, t, tailles_paquets,p) )
            if x != (False, []):
                return x
            #fin traitement
            i = 0
        while i<t and num[i]==tri_imax[i]:
```

```
                num[i] = 0
                i += 1
        return False, []

def isomorphism_tri(G,H,triG,triH):
    if G.nbr != H.nbr:
        return False, []
    if [len(e) for e in triH] != [len(e) for e in triG]:
        return False, []
    return recherche_isom(G, H, triG, triH)

#McKay

#partition des sommets sn -> partition equitable des sommets R(s)
    (cf mckay's canonical graph labeling)

def deg(G,w,V): #G un graphe, w un sommet de G, V une partie de G
    (partie de [|0,n-1|])
    degV = 0
    for x in G.connect[w]:
        if x in V:
            degV += 1
    return degV

def shatters(G,Vj,Vi): #Vj shatters Vi
    nvi = len(Vi)
    for k in range(nvi-1):
        for l in range(k+1,nvi):
            if deg(G,Vi[k],Vj) != deg (G,Vi[l],Vj):
                return True
    return False
```

```
def shattering(G,Vi,Vj):#shattering of Vi by Vj / on suppose
    shatters(G,Vj,Vi) / renvoie X=[X1,..,Xt] partition de Vi
    triés selon le degré dans Vj
    X = [[] for i in range(len(Vj)+1)] #au maximum, le degré d'un
        sommet de Vi dans Vj est len(Vj)
    for u in Vi:
        if deg(G,u,Vj) > len(Vj):
            print('u = ',u,' ; Vj = ',Vj,' ; deg = ' ,deg(G,u,Vj),
                ' ; len(Vj) = ', len(Vj))
            print([x for x in G.connect[u] if x in Vj])
        X[ deg(G,u,Vj) ].append(u)
    return [x for x in X if x != []]

def refinement(G,s):#s=[V0,V1,...] une partition de [|0,n-1|],
    renvoie R(s) une partition équitable, propager l'information
    du degré ( cf part.4 mckay's canonical graph labeling )
    Rs = s.copy()
    B = [(i,j) for i in range(len(Rs)) for j in range(len(Rs)) if
        shatters(G,Rs[j],Rs[i])]
    im, jm = 0, 0
    while B!=[]:
        im, jm = min_couples(B)
        Rs = Rs[:im] + shattering(G,Rs[im],Rs[jm]) + Rs[im+1:]
        B = [(i,j) for i in range(len(Rs)) for j in range(len(Rs))
            if shatters(G,Rs[j],Rs[i])]
    return Rs

#relation d'ordre sur les graphes, nombre binaire donné par l'
    ordre lexicographique des arêtes
```

```
def str_is(G,i,j):
    if j in G.connect[i] or i in G.connect[j]:
        return '1'
    return '0'

def i(G):#binary sequence of G
    bin = ''
    for i in range(G.nbr-1):
        for j in range(i+1,G.nbr):
            bin += str_is(G,i,j)
    return ''.join(bin)

def plus_grand_iG(iG,iH):
    if iG == '':
        return True
    elif iG[0]=='1' and iH=='0':
        return True
    elif iG[0]=='0' and iH=='1':
        return False
    else:
        plus_grand(iG[1:],iH[1:])


def plus_grand_v1(G,H):
    return plus_grand(i(G),i(H))

def plus_grand_v2(G,H):#i(G) et i(H) calculés au fur et à mesure,
      arrêt selon  ordre lexico
    for i in range(G.nbr-1):
        for j in range(i+1,G.nbr):
            g, h = int(str_is(G,i,j)), int(str_is(H,i,j))
```

```
            if g != h :
                return g and not h
    return True

def max_graphes(G, l_permut) :
    intn = [i for i in range(G.nbr)]
    maxi, sigma_max = G, l_permut[0]
    for sigma in l_permut :
        adj_permut = [[] for _ in range(G.nbr)]
        for i in range(G.nbr) :
            adj_permut[sigma[i]] = [sigma[e] for e in G.connect[i
                ]]
        G_temp = Graph(intn,adj_permut)
        if plus_grand_v2(G_temp,maxi) :
            maxi, sigma_max = G_temp, sigma
    return maxi, sigma_max


#search tree

class Tree :
  def __init__(self, val = None) :
    self.node = val
    self.list = None

def leaf(T) :#renvoie liste des feuilles
    l = []
    pile = [T] # parcours en profondeur
    while pile != [] :
        t = pile.pop()
        if t.list==None :
```

```
            l.append(t.node)
        else:
            for tt in t.list:
                pile.append(tt)
    return l

def first_part(s):#renvoie la première partie non triviale de la
     partition s
    for i in range(len(s)):
        if len(s[i])>1:
            return i
    return -1

def fils(G,t,affichage_arbre=False):#t un noeud
    s, u = t
    if len(s)==G.nbr:
        return None
    i = first_part(s)
    l_fils = []
    for ui in s[i]:
        R = refinement(G,s[:i] + [[ui],[x for x in s[i] if x!=ui]]
              + s[i+1:])
        if affichage_arbre:
            print((R, u+[ui]))
        l_fils.append( Tree((R, u+[ui])) )
    return l_fils

def incrementer(G,T):
    if T.list==None:
        f = fils(G,T.node)
        T.list = f
```

```
        return f!=None

    else:
        continu = False
        for t in T.list:
            if incrementer(G,t):
                continu = True
        return continu

def search_tree(G, s):#search tree dont la racine est ( s=(V1|V2
    |...), [] ), à chaque étage les fils sont (s perpend u) pour
    u dans V_i, V_i la première partie diff d'un singleton (cf
    mckay's...)
    T = Tree((refinement(G,s),[]))
    continu = True
    while continu:
        continu = incrementer(G,T)
    return T

def permutations_tree(G, s):
    l = leaf(search_tree(G,s))
    return [[singleton[0] for singleton in x[0]] for x in l]

def Cm(G,s=[]):
    if s==[]:
        s = [[i for i in range(G.nbr)]]
    l_permut = permutations_tree(G,s)
    return max_graphes(G, l_permut)

def isomorphes_mckay(G,H,sg=[],sh=[]):
    CmG, CmH = Cm(G,sg), Cm(H,sh)
```

```
    return (CmG[0].connect == CmH[0].connect), CmG[1], CmH[1]

def isomorphes_feuilles(G,H,sg=[],sh=[]):
    l_permut_G = permutations_tree(G,sg)
    l_permut_H = permutations_tree(H,sh)
    for i in range(len(l_permut_G)):
        if test_isomorphism2(G,H,l_permut_G[i],l_permut_G[i]):
            print( test_isomorphism2(G,H,l_permut_G[i],l_permut_G[
                i]) )
            return True, []
    return False, []
```

```
#prot -> graphe (perte d'infos)

def graph_of_prot(prot):
    return Graph([i for i in range(prot.nbr)], prot.connect)

#verifier le tri

def verif(G,tri):
    for i in range(G.nbr):
        if not (True in [(i in t) for t in tri]):
            return False
    return True

#tri des sommets pour une protéine

def sort_by_deg_atom(prot): #proposition d'un tri canonique par
    degre et type d'atome
```

```
    G = graph_of_prot(prot)
    ld = G.sort_by_degre()
    la = []
    for at in ['C','N','O','H','S']:
        l = []
        for i in range(prot.nbr):
            if prot.latom[i].atom == at:
                l.append(i)
        la.append( l )
    la = [l for l in la if l!=[]]
    return assoc_canonique([ld,la]) #on combine les
        caractéristiques degre et type

#test d'isomorphisme tri

def isom_invariants(prot1,prot2):
    if prot1.nbr==0 or prot2.nbr==0:
        return False, []
    G, H = graph_of_prot(prot1), graph_of_prot(prot2)
    triG, triH = refinement(G,sort_by_deg_atom(prot1)), refinement
        (H,sort_by_deg_atom(prot2))
    isom = isomorphism_tri(G, H, triG, triH)
    return isom

#test d'isomorphisme mckay

def isom_mckay(prot1,prot2):
    d = time()
    if prot1.nbr==0 or prot2.nbr==0:
        return False, 0
    G, H = graph_of_prot(prot1), graph_of_prot(prot2)
```

```
    sg, sh = sort_by_deg_atom(prot1), sort_by_deg_atom(prot2)
    isom = isomorphes_mckay(G,H,sg,sh)
    return isom, time()-d

def isom_mckay_feuilles(prot1,prot2):
    d = time()
    G, H = graph_of_prot(prot1), graph_of_prot(prot2)
    sg, sh = sort_by_deg_atom(prot1), sort_by_deg_atom(prot2)
    isom = isomorphes_feuilles(G,H,sg,sh)
    return isom, time()-d
```

```
def filtre(prot,li):
    latom = [Atom(li.index(k), prot.latom[k].point, prot.latom[k].
        atom) for k in li]
    connect = [[li.index(v) for v in inter2(li,prot.connect[s])]
        for s in li]
    return Protein(latom, connect)

def filtre_aretes(prot,lar):
    li = []
    for (i,j) in lar:
        if i not in li:
            li.append(i)
        if j not in li:
            li.append(j)
    latom, connect = [Atom(li.index(k), prot.latom[k].point, prot.
        latom[k].atom) for k in li], [[] for _ in li]
    for s in li:
        for v in prot.connect[s]:
```

```
            if ((s,v) in lar or (v,s) in lar):
                connect[li.index(s)].append(li.index(v))
    P = Protein(latom, connect)
    if len(connexe(P,True)) > 1:
        return Protein([],[])
    return P

def subgraph(pg,pH,lH,ng,nh): #pg une sous protéine de pG (
    attention pas les mêmes numérotations)
    partH = parties(nh,ng) #ensemble des parties à ng éléments de
        [|0,nh-1|]
    for part in partH:
        ph = filtre_aretes(pH, [lH[i] for i in part]) #restriction
            de pH
        if pH!=None:
            ism, sigma = isom_invariants(pg,ph) #test d'
                isomorphisme sur les 2 restrictions
            if ism:
                save(ph, 'ph')
                return True, [lH[i] for i in part]
    return False, []

def subgraph_aretes(pG,pH,lH,ng,nh,larg): #pG et pH les protéines
    completes
    s = filtre_aretes(pG,larg) #restriction de pG à la liste larg
        d'arêtes
    if s==None: #pas connexe -> en "plusieurs morceaux"
        return False, []
    return subgraph( s, pH, lH,ng,nh)

def search_sub(pG,pH):
```

```
    lG, lH = pG.liaisons_sans_doublons_indice(), pH.
        liaisons_sans_doublons_indice() #liste des liaisons
    ng, nh = len(lG), len(lH) #nombres de liaisons
    if pG.nbr > pH.nbr :
        return search_sub(pH,pG)
    for n in range(pG.nbr,0,-1) : #nombre de liaisons de pG
        sélectionnées
        print(n)
        partG = parties(ng,n) #parties de n éléments de [|0,ng-1|]
        for part in partG :
            b, s = subgraph_aretes(pG, pH, lH, n, nh, [lG[i] for i
                in part]) #conservation des n arêtes
                séléctionnées
            if b : #b si il existe une sous-partie de pH (de n
                arêtes) isomorphe à la protéine pG réduite
                save(filtre_aretes(pG,[lG[i] for i in part]),'pg')
                return True, s
    return False, []

def infos_sub(prot1,prot2) :
```

```
from extract_PDB import *

#gen_isom_prot(0.1,False)

#gen_isom_prot()

def infos_tris(prot1,prot2,afficher_permut=False):
    deps = time()
    s1, s2 = sort_by_deg_atom(prot1), sort_by_deg_atom(prot2)
    arrs = time()
    G1, G2 = graph_of_prot(prot1), graph_of_prot(prot2)
    deprs = time()
    rs1, rs2 = refinement(G1,s1),refinement(G2,s2)
    arrrs = time()
    m = max([len(l) for l in s1])
    print('s1')
    for i in range(1,m+1):
        e = sum([len(l)==i for l in s1])
        if e!=0:
            print('{} : '.format(i), e)
    print('')
    print('rs1')
    m= max([len(l) for l in rs1])
    for i in range(1,m+1):
        e = sum([len(l)==i for l in rs1])
        if e!=0:
            print('{} : '.format(i), e)
    print('')
    print('tri deg/atom : ',arrs-deps,' ; refinement : ',arrrs-
        deprs)
```

```
    tri_ex = isom_invariants(prot1,prot2)
    if afficher_permut:
        print(tri_ex[1])
    print('isomorphes : ',tri_ex)

def infos_mckay(prot1,prot2,afficher_permut=False):
    #mckay_ex, t2 = isom_mckay(prot_ex_isom1,prot_ex_isom2)
    mckay_ex, t2 = isom_mckay_feuilles(prot_ex_isom1,prot_ex_isom2
        )
    if afficher_permut:
        print(mckay_ex[1])
    print('')
    print('ismorphes ',mckay_ex[0], t2)
```

```
def infos_sub(prot1,prot2):
    d = time()
    sub_b, ls = search_sub(prot1,prot2)
    return sub_b, ls, time()-d
```

```
import numpy as np
import random as rd

#divers

def fact(n):
    k = 1
    for i in range(2,n+1):
        k = k*i
    return k

def permutations(n):#liste des permutations d'un ensemble
    [0,1,...,n-1]
    if n<=1:
        return [[0]]
    pm, l = permutations(n-1), []
    for i in range(len(pm)+1):
        for p in pm:
            l.append( p[:i] + [n-1] + p[i:] )
    return l

def permutliste(seq, er=False):#permutations non récursif, er=
    False si pas de répétition
    p = [seq]
    n = len(seq)
    for k in range(n):
        for i in range(len(p)):
            z = p[i][:]
            for c in range(n-k):
                z.append(z.pop(k))
```

```
                print(z)
                if er==False or (z not in p):
                    p.append(z[:])
    return p

def permut_dynamique(n):#renvoie la liste des permutation de 0:[] à n
    permut = [ [] , [[0]], ]
    for k in range(2,n+2):
        l  = []
        lm = len(permut[k-1])
        for i in range(lm+1):
            for p in permut[k-1]:
                l.append( p[:i] + [k-1] + p[i:] )
        permut.append(l)
    return permut

def permut_dyn_liste(p, l):#p = permut_dynamique(n) avec len(l)<=n
    k = len(l)
    return [[l[i] for i in f] for f in p[k]] #on applique la permutation à la liste

def indice_min(l, value=False):
    n = len(l)
    mini, i0 = l[0], 0
    for i in range(n):
        if l[i] < mini:
            mini, i0 = l[i], i
    if value:
        return i0, mini
```

```
    return i0

def indice_max(l, value=False):
    n = len(l)
    maxi, i0 = l[0], 0
    for i in range(n):
        if l[i] > maxi:
            maxi, i0 = l[i], i
    if value:
        return i0, maxi
    return i0

def privede(l1, l2):
    return [x for x in l1 if not x in l2]

def numerotation(l):
    t, num, i = len(l), [-1]+[0]*(t-1), 0
    while i<t:
        if num[i]<l[i]:
            num[i] += 1
            #traitement
            i = 0
        while i<t and num[i]==l[i]:
            num[i] = 0
            i += 1

#parties

def parties(n,k):#liste des parties à k éléments de [|0,n-1|] (
    même ordre)
```

```
    num, i, s, part = [-1]+[0]*(n-1), 0, -1, [] #s est la somme de
        num
    while i<n :
        if num[i]<1 :
            num[i] += 1
            s += 1
            if s==k :
                part.append( [i for i in range(n) if num[i]] )
            i = 0
        while i<n and num[i]==1 :
            num[i] = 0
            s+= -1
            i += 1
    return part

def sans_repet(l) :
    return [l[i] for i in range(len(l)) if not (l[i] in l[i+1:])]

def sans_repet_tri(l) :
    if l==[] :
        return []
    n = max(l)+1
    lind = [False]*n
    for x in l :
        lind[x] = True
    return [i for i in range(n) if lind[i]]

def doublon(l) : #bool, unicité des termes de l
    for i in range(len(l)-1) :
        if l[i] in l[i+1:] :
            print(l[i], l[i+1:])
```

```
                return True
        return False

def min_couples(lcouples):#ordre lexicographique / non vide
        def plus_petit(e,f):
                (a,b),(c,d) = e,f
                return a<c or (a==c and b<=d)
        cmin = lcouples[0]
        for x in lcouples:
                if plus_petit(x,cmin):
                        cmin = x
        return cmin

#opérations sur les permutations

def inverse(sigma):
        n = len(sigma)
        sigma_inv = [0]*n
        for i in range(n):
                sigma_inv[sigma[i]] = i
        return sigma_inv

def composee(sigma2,sigma1):
        return [ sigma2[sig1] for sig1 in sigma1 ]

#geometrie en 3 dimensions

def distance(A,B):
        return np.sqrt( abs(A[0]-B[0])**2 + abs(A[1]-B[1])**2 + abs(A
                [2]-B[2])**2 )
```

```
def angle(ab,ac,bc):#angle en B en radians
    x = (ab**2 + bc**2 - ac**2) / (2*ab*bc)
    return np.arccos(x)

def orientertriangle(A,B,C,ld):#renvoie les points dans l'ordre
     base,coté1,cote2 et ld=[dab,dac,dbc]
    dmax, i0 = 0, 0
    if ld[0]>ld[1] and ld[0]>ld[2]:
        pass
    elif ld[1]>ld[0] and ld[1]>ld[2]:
        i0 = 1
    else:
        i0 = 2
    return [(A,B,C), (A,C,B), (B,C,A)][i0], [ ld, [ld[1],ld[2],ld
         [0]], [ld[2],ld[0],ld[1]] ][i0]

def airetriangle(A,B,C):
    (A,B,C), ld = orientertriangle( A,B,C, [distance(A,B),distance
         (A,C),distance(B,C)] )
    ab, ac, bc = tuple(ld)
    alpha = angle(ab,ac,bc)
    h = bc * np.sin(alpha)
    base = ab
    return 0.5 * base * h

def airepoly(A,B,C,D):
    return airetriangle(A,B,C) + airetriangle(D,B,C)

def matrice_dist(l):#liste de coordonnées
    n = len(l)
    mat = [[0]*n for _ in range(n)]
```

```
    for i in range(n):
        for j in range(i+1,n):
            mat[i][j] = distance(l[i],l[j])
            mat[j][i] = mat[i][j]
    return mat

def alignement(ssp1,ssp2):#2 protéines supposées isomorphes,
     renumérotées selon la permutation, de même sommets
    N1, N2 = ssp1.liste, ssp2.liste
    offset_moy = [0,0,0]
    for i in range(ssp1.nbr):
        offset_moy = [offset_moy[k] + (1/ssp1.nbr)*(N2[i][k] - N1[
            i][k]) for k in [0,1,2]]
    for i in range(ssp1.nbr):
        N2[i] = [ N2[i][k]-offset_moy[k] for k in [0,1,2] ]
    return nuagepoints(N1), nuagepoints(N2)

#nuage_de_point

def distpoints(NA,NB):
    return [distance(NA.liste[i],NB.liste[i]) for i in range(NA.
        nbr)]

def dist_reset(NA,NB):
    D = [distance(NA.liste[0],NB.liste[0])]
    vecta = np.array([0.,0.,0.])
    vectb = np.array([0.,0.,0.])
    for i in range(NA.nbr-1):
        vecta += np.array(NA.liste[i+1]) - np.array(NA.liste[i])
        vectb += np.array(NB.liste[i+1]) - np.array(NB.liste[i])
```

```
        D.append( distance(np.array(NA.liste[i+1])-vecta, np.array
            (NB.liste[i+1])-vectb) )
    return D

def liste_angles(NA,NB):
    theta = []
    for i in range(NA.nbr-1):
        vect = np.array(NA.liste[i]) - np.array(NB.liste[i])
        new_A = np.array(NB.liste[i+1]) + vect
        d1 = distance( new_A, NA.liste[i])
        d2 = distance( new_A, NA.liste[i+1])
        d3 = distance( NA.liste[i], NA.liste[i+1] )
        theta.append( angle(d1,d2,d3) )
    return theta

def aires_liste(NA,NB):
    A = []
    for i in range(NA.nbr - 1):
        A.append( airepoly(NA.liste[i+1],NB.liste[i+1],NA.liste[i
            ],NB.liste[i]) )
    return A

def aire(NA,NB):
    return sum( aires_liste(NA,NB) )

def color_dlines(dl): # entre 1 et 10
    n, m, M = len(dl), min(dl), max(dl)
    lcolor = [0 for _ in range(n)]
    for i in range(n):
        if dl[i]==m:
            lcolor[i] = 1
```

```
        else :
            lcolor[i] = int( np.ceil(10*(dl[i]-m) / (M-m)) ) - 1
    return lcolor

def rotation_z(liste,x=5,y=5,theta=-np.pi/2):
    r = np.array([[np.cos(theta), np.sin(theta),0], [np.sin(theta)
        , np.cos(theta),0], [0,0,1]])
    ref = np.array([x, y, 0])
    def torefcolonne(i):
        return (np.array(i)-ref).reshape(3,1)
    N1 = [(ref + torefcolonne(point).reshape(1,3)).tolist()[0] for
        point in liste]
    return N1

def paramtriangle(u,v,A,B,C):
    O = np.array(A)
    v1 = np.array(B) - np.array(A)
    v2 = np.array(C) - np.array(A)
    if v <= (1-u):
        return O + u * v1 + v * v2
    return O + v2

#stats

def esperance(l):
    return sum(l)/len(l)

def variance(l):
    print(l)
    e1 = esperance([x**2 for x in l])
    e2 = esperance(l)**2
```

```
    return e1 - e2

def ecart_type(l):
    return np.sqrt(variance(l))
```

```
#Coefficients

def r_coeff1(NA,NB) :
    return 1 / ( 1 + sum(distpoints(NA,NB))/NA.len ) #creuser
        ecart avec 1 -> passer sum... à une racine nieme

def r_coeff2(NA,NB) :
    return 1 - sum(distpoints(NA,NB))/NA.len

def r_coeff3(NA,NB) :
    return 1 / ( 1 + aire(NA,NB) / NA.len**2 )

def r_coeff4(NA,NB) :
    return 1 / ( 1 + np.sqrt(aire(NA,NB)) / NA.len )

def r_coeff5(NA,NB) :
    return (r_coeff4(NA,NB))**2

def r_coeff6(NA,NB) :
    return (1 / ( 1 + sum(dist_reset(NA,NB))/NA.len) )

def r_coeff_dist(NA,NB) :
    return  1/(1+ (sum([ abs(NA.dist[i] - NB.dist[i]) for i in
        range(NA.nbr - 1) ]) / max(NA.len,NB.len)))

def r_coeff_angles(NA,NB) :
    theta = liste_angles(NA,NB)
    return 1/(1+(sum([ abs(np.sin(theta[i]/2)) * max(NA.dist[i],NB
        .dist[i]) for i in range(NA.nbr-1)]) / sum([max(NA.dist[i
        ],NB.dist[i]) for i in range(NA.nbr-1)])))
```

```
def r_coeff7(NA,NB,k=2):
    return ( r_coeff_dist(NA,NB) + r_coeff_angles(NA,NB) ) / 2
```

```
#coeff

from cas_simple_nuage import *

def coeff_prox(ssp1,ssp2):#2 protéines supposées isomorphes,
     renumérotées selon la permutation, de même sommets
    N1, N2 = alignement(ssp1,ssp2)
    return 100 * r_coeff7(N1,N2) #moyenne du coefficient des
         distances et angles

def coeff_struct(pG,pH,ssp):
    ng, nh, np = len(pG.liaisons_sans_doublons_indice()), len(pH.
         liaisons_sans_doublons_indice()), len(ssp.
         liaisons_sans_doublons_indice())
    print(ng, nh, np)
    return int(100*(2*np)/(ng+nh))

def coeff_tot(pG,pH):
    b, r = search_sub(pG,pH)
    if not b:
        return 0
    spg, sph = recup('pG'), recup('pH')
    cp, cs = coeff_prox(spg, sph), coeff_struct(pG,pH,spg)
    print(cp, cs)
    variance = (100 - cs)/2
```

```
    c = cs - variance + 2*variance*(cp/100)
    return c

p1 = recup('lim_22_1')
p2 = recup('lim_22_2')

print(coeff_tot(p1,p2))
```

```
import matplotlib.pyplot as plt
from geometrie_et_aux import fact
from numpy import log10

def brute(n,q):
    return sum([log10(x) for x in range(1,n*q)])

def brute_tri(n,q):
    return n * sum([log10(x) for x in range(1,q)])

QC = [(2,'b'),(3,'g'),(5,'r'),(7,'purple')]
N = [n for n in range(1,51)]

fig, ax = plt.subplots()
for (q,c) in QC:
    B = [brute(n,q) for n in N]
    BT = [brute_tri(n,q) for n in N]
    plt.plot([0]+N, [0]+B, color =c, label = 'brute avec q = {} '.
        format(q))
    plt.plot([0]+N, [0]+BT, color = c, ls = '--', label = ' tri
        avec q = {} '.format(q))
plt.ylim(0,300)
plt.xlabel("n, taille de G et H", loc = 'right')
plt.ylabel(" C(n) ", loc = 'top')
plt.title(" Complexité par force brute (nq éléments) et par tri
    des sommets (n paquets de taille q)")

ticks = ax.get_yticklabels()
print(ticks)
posticks =ax.get_yticks()
```

```
for i in range(len(posticks)):
    ticks[i].set_text('10^'+str(int(posticks[i]))+' ')
    ticks[i].set_usetex(True)
ax.set_yticks(posticks)
ax.set_yticklabels(ticks)
plt.legend()
plt.show()
```

```
from manim import *
from cas_simple_nuage import *

N = rot_z(genererliaisonsunif(10,10)) #10 points
#M = rot_z(genererliaisonsunif(10,10))
M = rot_z(decalagex(N,2,0))

colorGOR = [color.Color(i) for i in ['#86f686', '#58d658', '#28
    ad28', '#ffff58', '#ffe500', '#ffc65c', '#f19c00', '#ff6464',
     '#ff4848', '#e70000']]
color_dist = color_dlines( distpoints(N,M) )

def no(l):
    a,b,c = l[0], l[1], l[2]
    return [a-5, b-5, c]

class DLines(ThreeDScene):
    def construct(self):

        axes = ThreeDAxes((0, 10), (0, 10), (0, 10), 10,10,10)
```

```
        #limits = tuple([Dot3D(point=no(x), color = YELLOW) for x
            in [[10,0,0],[0,10,0],[0,0,10]] ])
        #dn = tuple( [Dot3D(point=no(x), color = BLUE) for x in N.
            liste] )
        #dm = tuple( [Dot3D(point=no(x), color = PURPLE) for x in
            M.liste] )
        ln = tuple( [Line3D(start=no(N.liste[i]), end=no(N.liste[i
            +1]), color=WHITE, thickness=0.05) for i in range(N.
            nbr-1)] )
        lm = tuple( [Line3D(start=no(M.liste[i]), end=no(M.liste[i
            +1]), color=GREEN_B, thickness=0.05) for i in range(N.
            nbr-1)] )
        #lines = tuple([Line3D(start=no(N.liste[i]), end=no(M.
            liste[i]), color= colorGOR[ color_dist[i] ]) for i in
             range(N.nbr)])

        self.add(axes, *lm, *ln)#, *lines) #*dn, *dm,  *limits)
        self.set_camera_orientation( phi=PI/3, theta=PI/6,
            frame_center=(0,0,5) )
        self.camera.set_zoom(0.4)

        self.begin_ambient_camera_rotation(rate=PI/4)
        self.wait(3)
        self.stop_ambient_camera_rotation()

#cd "C:\Users\Adrien Dubois\Desktop\TIPE\cas_simple"
#manim -spqk plot_distlines.py
#manim -pqh plot_distlines.py --disable_caching
```

```
from manim import *
from extract_PDB import *

def colormol(molecule):
    if molecule=='C':
        return RED
    if molecule=='H':
        return WHITE
    if molecule=='O':
        return BLUE
    if molecule=='N':
        return GREEN
    if molecule=='S':
        return ORANGE
    else:
        return black

def no(l):
    a,b,c = l[0], l[1], l[2]
    return [a/2,b/2,c/2+5] #[a/30-7, b/30-7, c/30]

prot = recup('lim_22_test')


class Protein(ThreeDScene):
    def construct(self):

        #axes = ThreeDAxes((0, 10), (0, 10), (0, 10), 10,10,10)
        ld = [Dot3D(point=no(molec.point), color = colormol(molec.
            atom)) for molec in prot.latom]
```

```
        print("Nombre d'atomes = ", len(ld))
        d = tuple( ld )
        lines = tuple([Line3D(start=no(x), end=no(y), color=GREY)
            for (x,y) in prot.liaisons_sans_doublons() ])
        self.add(*d, *lines)

        self.set_camera_orientation( phi=PI/3, theta=PI/3,
            frame_center=(0,0,5) )
        self.camera.set_zoom(0.6)

#           self.begin_ambient_camera_rotation(rate=PI/4)
#           self.wait(5.3)
#           self.stop_ambient_camera_rotation()

#cd "C:\Users\Adrien Dubois\Desktop\TIPE\2-code"
#manim -spqk plot_protein.py
```

```
from manim import *
from subisom_prot import *

def colormol(molecule):
    if molecule=='C':
        return RED
    if molecule=='H':
        return WHITE
    if molecule=='O':
        return BLUE
    if molecule=='N':
        return GREEN
```

```
    if molecule=='S':
        return ORANGE
    else:
        return black

prot1 = recup('lim14')
prot2 = recup('lim10')

offset1 = [min([at.point[i] for at in prot1.latom]) for i in
    [0,1,2]]

def no(l,offset=offset1):
    a,b,c = l[0]-offset[0], l[1]-offset[1], l[2]-offset[2]
    return [a-5,b-5,c] #[a/30-7, b/30-7, c/30] #c/2+5

class Protein(ThreeDScene):
    def construct(self):

        axes = ThreeDAxes((0, 10), (0, 10), (0, 10), 10,10,10)
        ld1 = [Dot3D(point=no(molec.point), color = colormol(molec
            .atom)) for molec in prot1.latom]
        ld2 = [Dot3D(point=no(molec.point), color = colormol(molec
            .atom)) for molec in prot2.latom]
        lines1 = tuple([Line3D(start=no(x), end=no(y), thickness
            =0.015, color=GREEN) for (x,y) in prot1.
            liaisons_sans_doublons() ])
        lines2 = tuple([Line3D(start=no(x), end=no(y), thickness
            =0.015, color=ORANGE) for (x,y) in prot2.
            liaisons_sans_doublons() ])
        self.add(*axes,*ld1,*ld2,*lines1,*lines2)
```

```
        self.set_camera_orientation( phi=3*PI/8, theta=PI/6,
             frame_center=(0,0,5) )
        self.camera.set_zoom(0.7)

#           self.begin_ambient_camera_rotation(rate=PI/4)
#           self.wait(5.3)
#           self.stop_ambient_camera_rotation()

#cd "C:\Users\Adrien Dubois\Desktop\TIPE\2-code"
#manim -spqk plot_sub.py
```