Échanges entre des piles de cartes

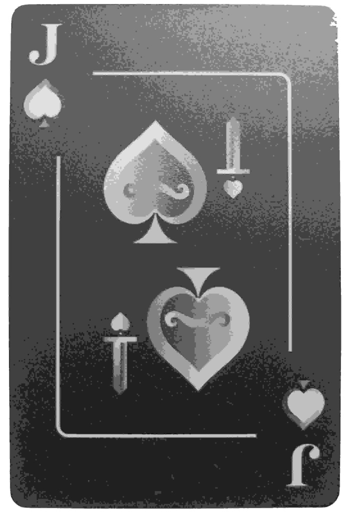
Sylvain WOZNY - 46196

MP2I

2023-2024

Introduction

Le jeu du Pouilleux (simplifié)



Qui a le plus de chance d'avoir le Valet de Pique après quelques tours ?
Et après beaucoup de tours ?
Et suivant le nombres de joueurs ?

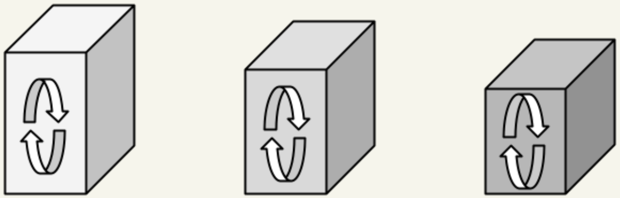
Problématique

Problématique : "Comment représenter l'évolution de la composition de plusieurs piles de cartes au cours du temps ? "

Plan

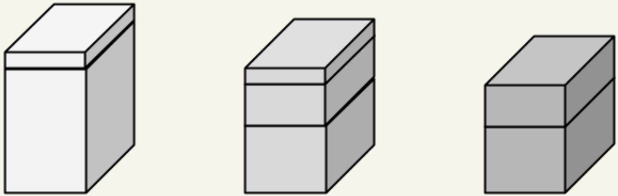
Généralité

Méthode d'échange - Mélange



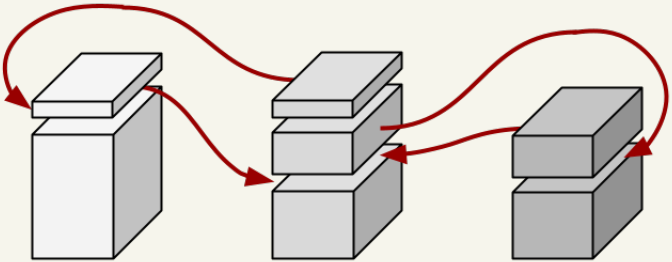
Généralité

Méthode d'échange - Paquets



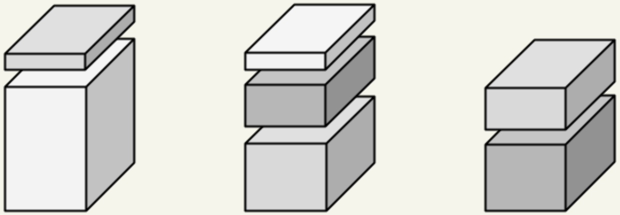
Généralité

Méthode d'échange - Envois

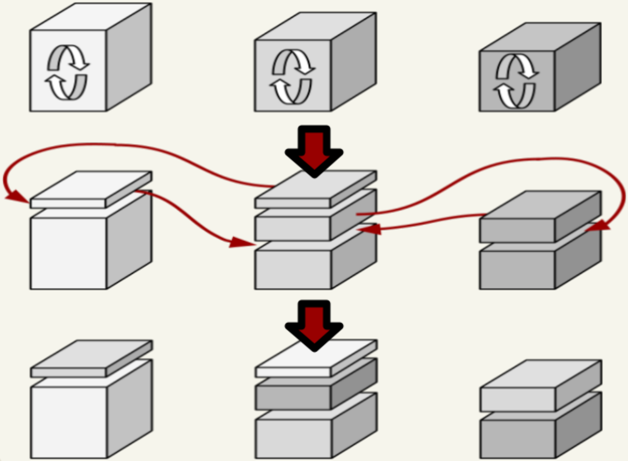


Généralité

Méthode d'échange - Envois

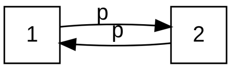


Mélange -> Paquets -> Envois

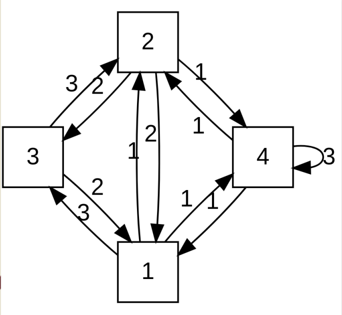


Représentation de la table : Graphe
Représentation d'une pile : Sommet
Représentation d'un échange entre une pile A et B : Arcs reliant A à B.

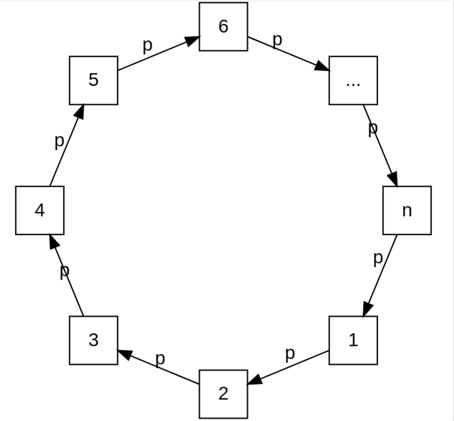
Graphe Bi-pile, Réciproque, Paquets égaux



Graphe "Équilibré"



Graphe n-piles, Cyclique, Paquets égaux



```
digraph finite_state_machine {
    rankdir=LR;
    node [shape = square]; 1 2 3 4 5 6 7 8
    #1 -> 1 [label = ""];
    1 -> 2 [label = "1"];
    #1 -> 3 [label = ""];
    1 -> 4 [label = "4"];
    1 -> 5 [label = "10"];
    1 -> 6 [label = "5"];
    1 -> 7 [label = "2"];
    #1 -> 8 [label = ""];
...
```

Matrice équilibrée :

Matrice carrée R de taille n

Condition sur les coefficients :

$\forall i \in [|1, n|], \sum_{j=1}^{n} r_{i,j} = \sum_{j=1}^{n} r_{j,i}$

<=>

Graphe équilibré :

Pour tout sommet S alors,

$deg_{sortant}(S) = deg_{entrant}(S)$

Dans un graphe équilibré : Les composantes connexes sont fortement connexes.

Matrice stable :

Matrice carrée R de taille n

Condition sur les coefficients :

$$\forall i \in [|1, n|], \sum_{j=1}^{n} r_{i,j} < \text{Nombre de cartes dans la i-ème pile}$$

On considère les matrices stables et équilibrées dans la suite.

Matrices équilibrées :

$R_2 \in \mathbb{M}_{2,2}$

$$R_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

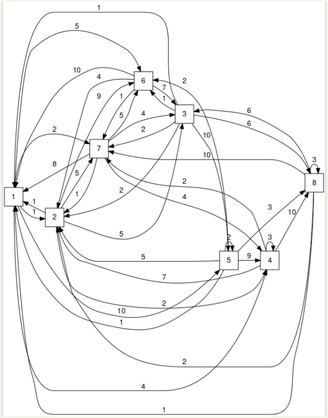
$R_8 \in \mathbb{M}_{8,8}$

$$R_8 = \begin{bmatrix} 0 & 1 & 0 & 4 & 10 & 5 & 2 & 0 \\ 1 & 0 & 5 & 2 & 0 & 9 & 5 & 0 \\ 1 & 2 & 0 & 0 & 10 & 1 & 2 & 6 \\ 0 & 7 & 0 & 3 & 0 & 0 & 2 & 10 \\ 1 & 5 & 0 & 9 & 2 & 2 & 0 & 3 \\ 10 & 4 & 7 & 0 & 0 & 0 & 1 & 0 \\ 8 & 1 & 4 & 4 & 0 & 5 & 0 & 0 \\ 1 & 2 & 6 & 0 & 0 & 0 & 10 & 3 \end{bmatrix} \text{ Équilibre à } 22$$

Généralité

Matrice Équilibrée/Stable

R_8 donne ce graphe :



D'où l'intérêt de la représentation matricielle.

Cas simple à 2 types de cartes : Noir/Rouge.

On s'intéresse aux cartes rouges

Carte = Booléen

```
carte1 = True #La carte 1 est rouge
carte2 = False #La carte 2 est noire
-
```

Pile = Liste de booléens

```
pile1 = [True,False,False,True,True]
```

Abscisse : n° d'étape

Ordonnée : Quotient de proportion en carte rouge dans la i-ème pile :

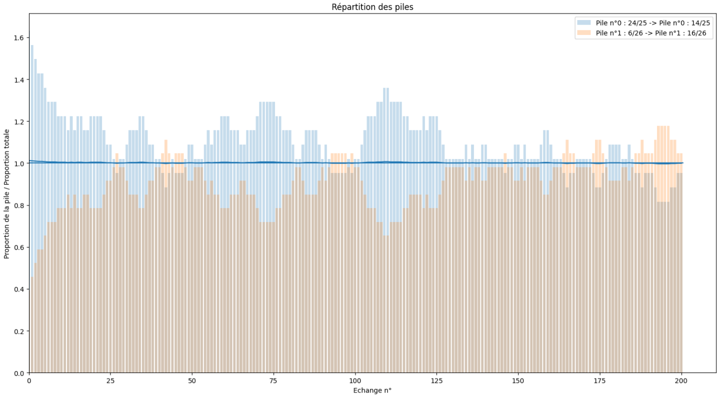
$$\frac{\text{proportion de carte touge dans la i-ème pile}}{\text{proportion de carte rouge sur la table}}$$

On superpose les histogrammes de toutes les piles (rendues transparentes).

Modélisation

Résultats

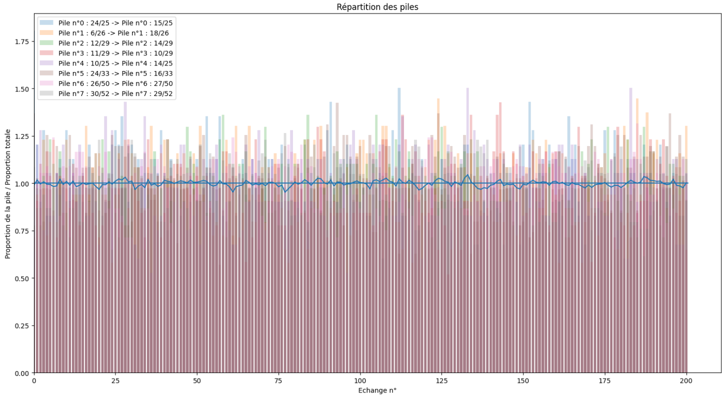
Pour R_2 :



Modélisation

Résultats

Pour R_8 :



Modélisation

Transition avec la Simulation

Problème de la modélisation :

- Instabilité
- Dépend de la méthode de mélange

Solution :

- Ne plus utiliser d'aléatoire
  - -> Utiliser des probabilités
- Ne plus plus stocker discrètement les cartes
  - -> Quantité stockée par des valeurs dans $\mathbb{R}$

Désavantage :

- Éloignement du réel
  - -> La pile contient $5,5$ cartes rouges.

t types de cartes
p piles de cartes

-

Représenter une pile :
Quantité de chaque type
OU
Proportion des types **ET** nombre total de cartes

-

Conservation de la matrice d'adjacence (à coefficients entiers).

-

Représentation d'un paquet de taille m provenant de la pile A :
Même proportion que la pile A et de taille m

[Première Simulation Testée]

$t = 2$

Pile $=$ Nombre de cartes noires $+ i$Nombre de cartes rouges

-

On obtient cette relation de récurrence :

$K' = RC \star (\Theta(K)(i-1) + C) + R^T(\Theta(K)(i-1) + C)$

avec :

$K =$Matrice de la table (des piles)

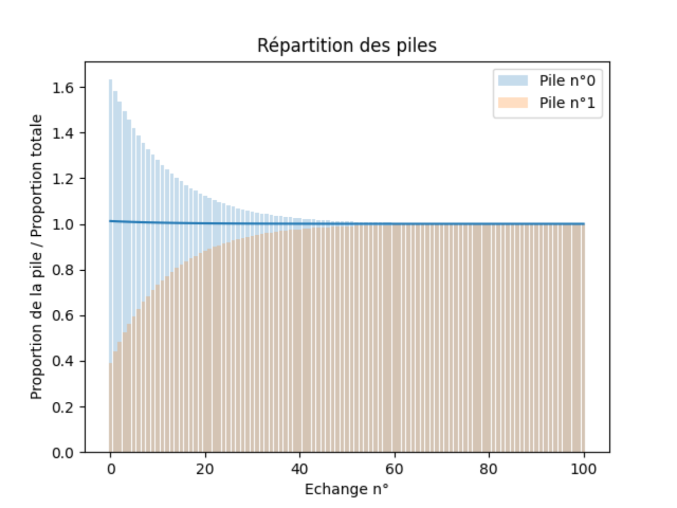
$R =$Matrice d'adjacence

$C =$ Matrice colonne de 1

$\Theta(K) = (\frac{\Im(k_i)}{k_i})_{i \in [|1,p|]}$

$\star =$ le produit d'Hadamard (terme à terme)

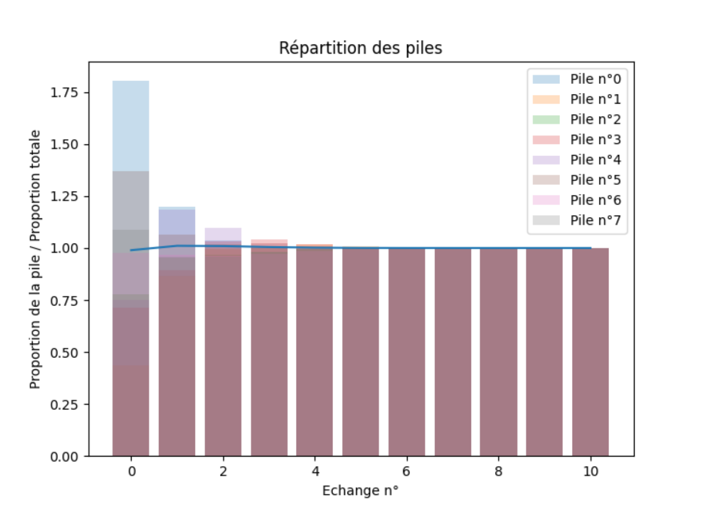
Pour R_2 :


Répartition des piles
Pile n°0
Pile n°1
Proportion de la pile / Proportion totale
Echange n°
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
20
40
60
80
100

Simulation

Résultats

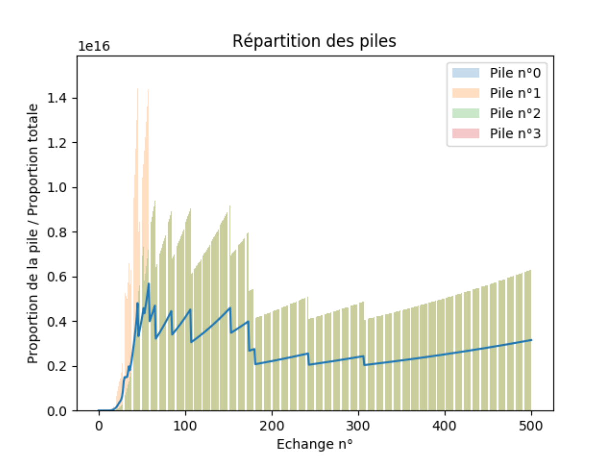
Pour R_8 :



Simulation

Importance de la stabilité et de l'équilibre

Pour une matrice non équilibrée et non stable de dimension 4 :



Simulation

Multi-type

Problèmes des histogrammes :

- Ne cible qu'un type de carte
- difficilement lisible pour $p > 2$

Solutions :

- Ajouter une nouvelle dimension : le temps
- Changer la transparence en empilement

Simulation

Représentation gif

Abscisse : n° de pile

Ordonnée : nombre de cartes

Couleur : type de carte

Titre : n° d'étape

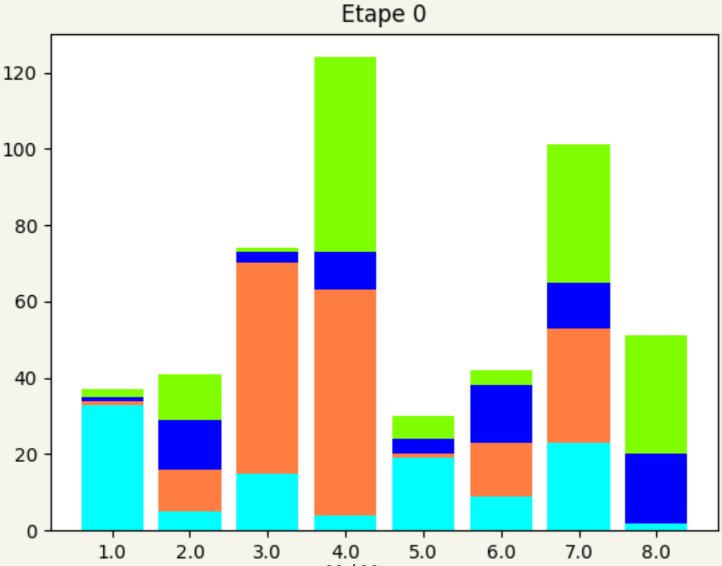
Chaque barre représente les cartes d'une pile, la proportion de couleur correspond à la proportion d'un type de carte dans la pile.

Évolution du gif au cours temps = Évolution au fil des étapes

*[Utilisation de la librairie python **imageio** (couplée à **matplotlib**]*

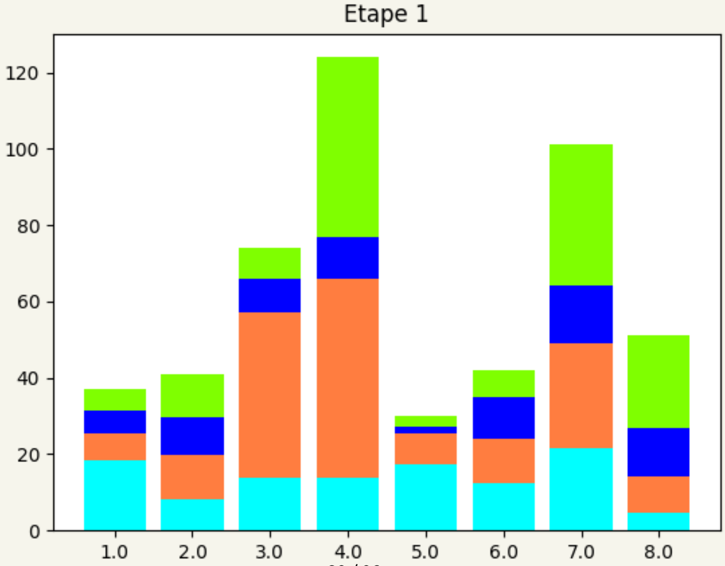
Simulation

Forme des résultats



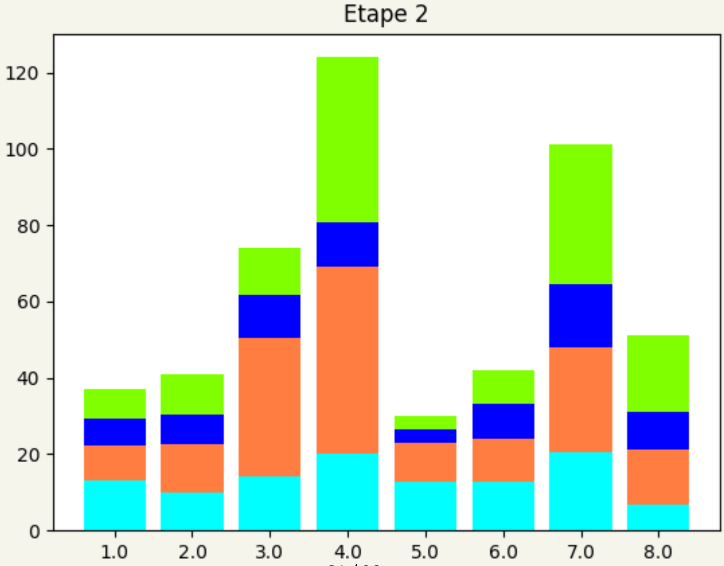
Simulation

Forme des résultats



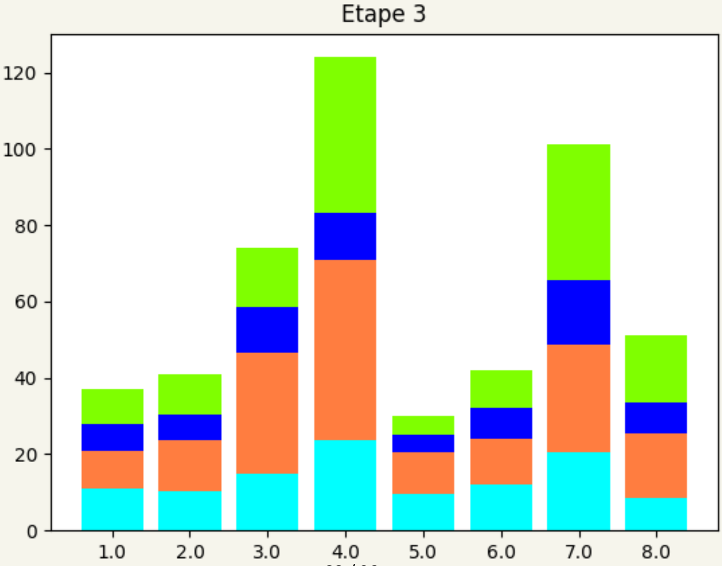
Simulation

Forme des résultats



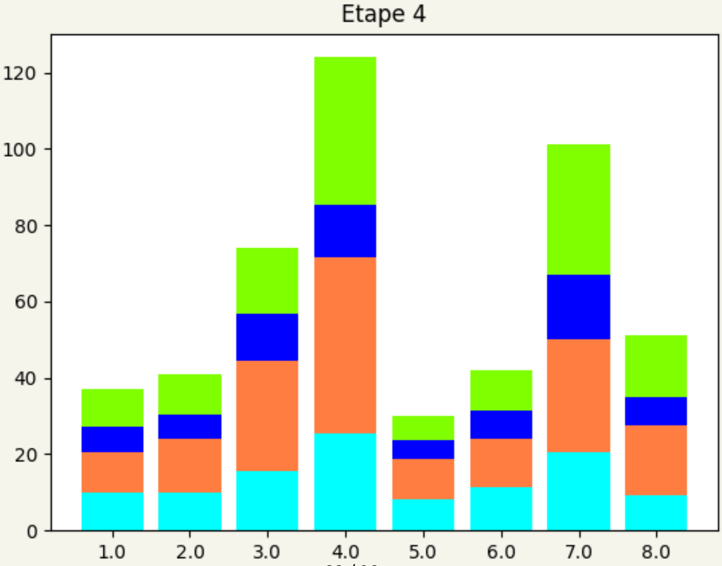
Simulation

Forme des résultats



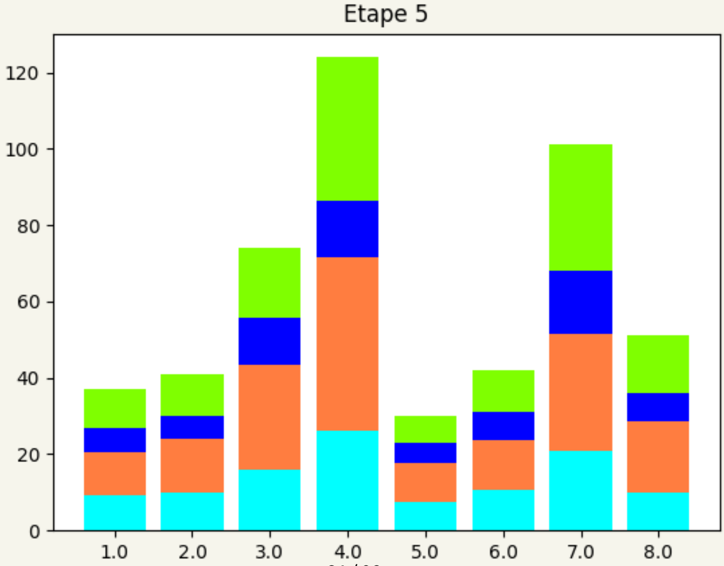
Simulation

Forme des résultats



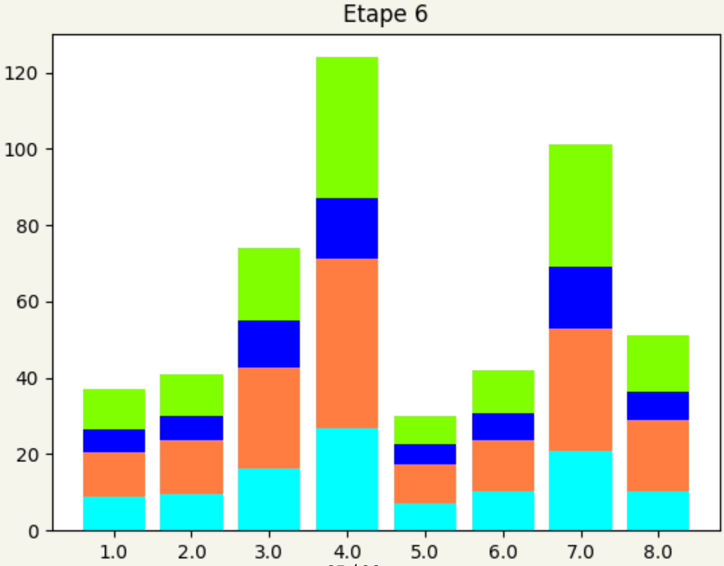
Simulation

Forme des résultats



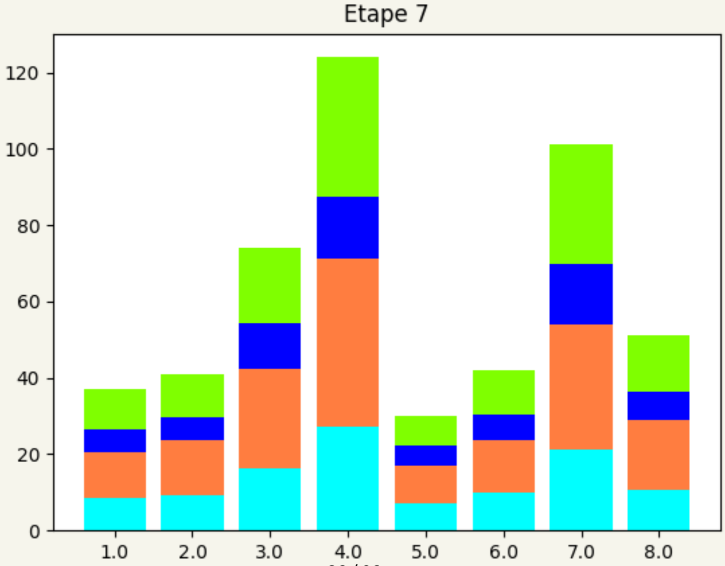
Simulation

Forme des résultats



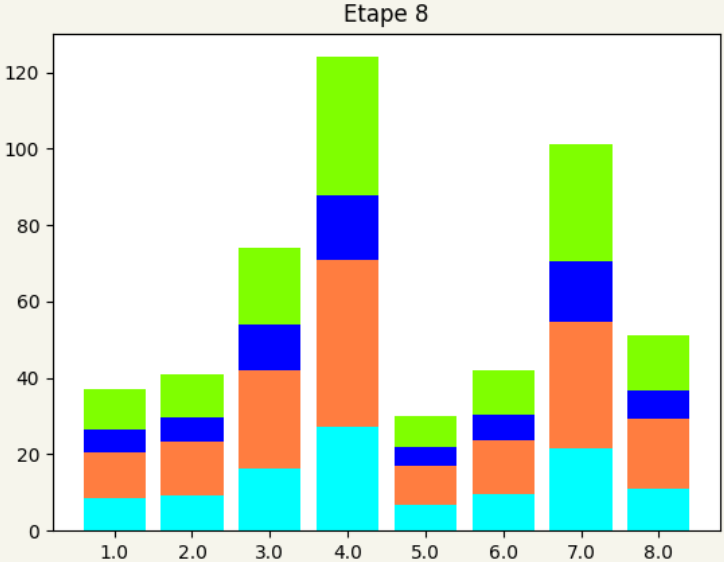
Simulation

Forme des résultats



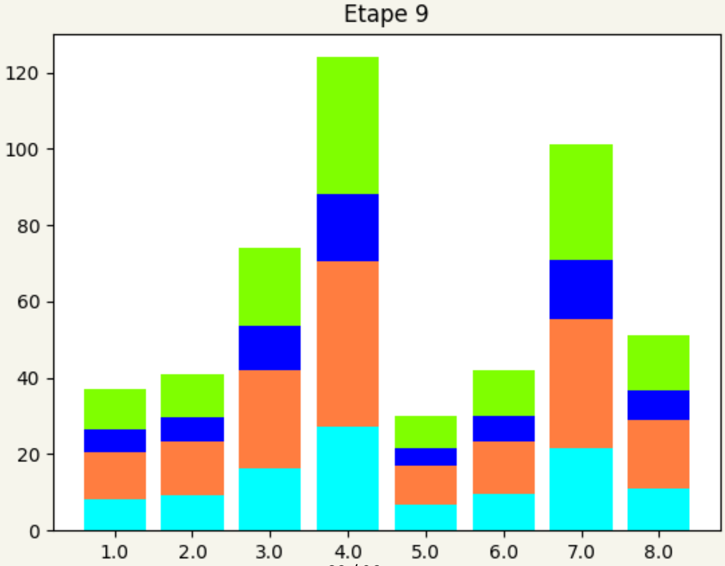
Simulation

Forme des résultats



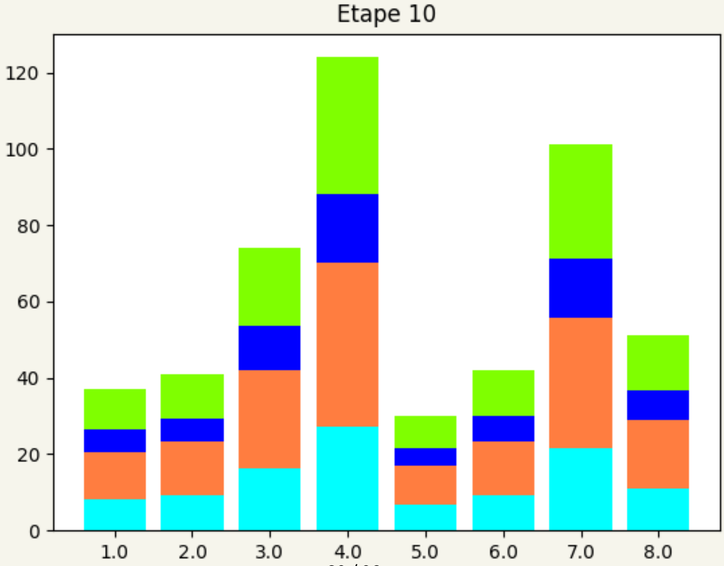
Simulation

Forme des résultats



Simulation

Forme des résultats



Représentations Multi-type :

Matrice d'adjacence : $R \in M_p(\mathbb{N})$

Matrice des quantités : $Q \in M_{p,1}(\mathbb{N})$ (Constante)

Matrice des proportions : $P \in M_{p,t}([0,1])$

Table = (P,Q)

Récurrence : $p'_{ij} = \frac{1}{q_i}(p_{ij}(q_i - \sum_{k=1}^{p} r_{ik}) + \sum_{k=1}^{p} p_{kj} r_{ki})$

Simulation

Transition Méthode Brute - Méthode de Markov

Problèmes :

- Récurrence compliquée
- Stockage en 2 matrices

Solutions :

- Trouver une représentation en une matrice
- Trouver une transition pratique entre les étapes

État de la table : $X = [pile_1, pile_2, ..., pile_p]$
Exemple :

```
X = [[20,30,10],[10,10,20],[30,0,10],[20,20,10]]
```

-

Matrice de Markov : Q dépend de R et X_0.

Éviter les imprécisions de calculs dues aux flottants :
$Q = \dfrac{1}{N_{max}} N_{max} Q$
avec $N_{max} = \prod_{i=1}^{p} taille_{pile_i}$ ainsi $N_{max} Q \in M_p(\mathbb{N})$

$$N = [taille_{pile_1}, \ldots, taille_{pile_p}]$$

$$Q = \begin{bmatrix} 1 - \frac{\sum_{j=2}^{p} R_{1,j}}{N_1} & \frac{R_{1,2}}{N_1} & \ldots & \ldots & \frac{R_{1,p}}{N_1} \\ \frac{R_{2,1}}{N_2} & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \frac{R_{p-1,p}}{N_{p-1}} \\ \frac{R_{p,1}}{N_p} & \ldots & \ldots & \frac{R_{p,p-1}}{N_p} & 1 - \frac{\sum_{j=1}^{p-1} R_{p,j}}{N_p} \end{bmatrix}$$

On a : $Q^n = \frac{1}{(N_{max})^n}(N_{max}Q)^n$.

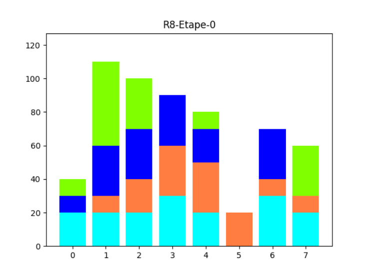
Après n étapes :
$X_n = X_{n-1}Q = X_0 Q \ldots Q = X_0 Q^n$

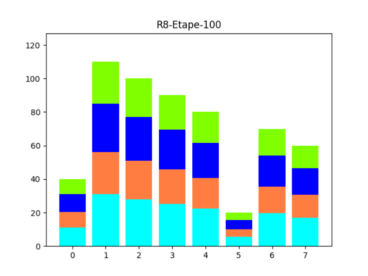
Attention : *numpy* ne permet pas facilement d'utiliser des entiers de tailles infinies dans des matrices. D'où l'utilisation de liste simple.

Simulation

Résultats

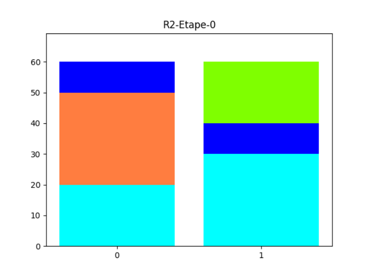
Pour R_8 :

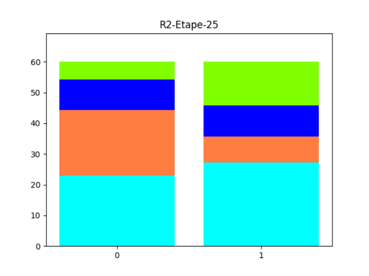




Simulation
Résultats

Pour R_2 :





Objectif :

- Trouver la disposition vers laquelle la table converge
- Trouver en combien d'étapes

Méthode :

- Trouver les composantes connexes (Kosaraju-Sharir)
- Pour chaque étape répartir les piles en familles (grâce à la Classification Hiérarchique Ascendante)
- Trouver la première étape avec égalité composantes connexes/familles

Recherche d'équilibre

Composantes connexes

Graphe associé à la matrice d'adjacence R : $G(R)$

Composantes connexes de $G(R)$: $C(R)$

Composantes fortement connexes de $G(R)$: $C_f(R)$

Soit $\mathcal{C} \in C_f(R)$, toutes les piles dans $\mathcal{C}$ convergent vers la même proportion de chaque type.

Que se passe-t-il pour des composantes connexes non fortement connexes ?
->Il n'y en a pas dans une matrice équilibrée, $C_f(R) = C(R)$

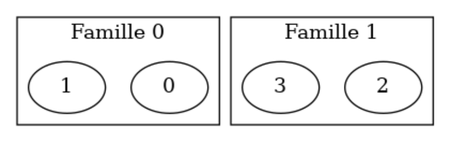
Comment chercher les composantes connexes ?
-> Algorithme de Kosaraju-Sharir

Recherche d'équilibre

Affichage des composantes via Graphviz par Python

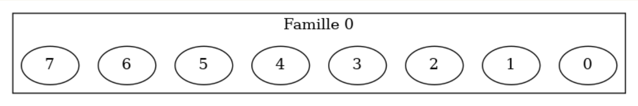
Affichage du graphe des composantes (fortement) connexes :

$$\begin{bmatrix} 0 & 5 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 \\ 0 & 0 & 6 & 0 \end{bmatrix} \quad => $$



.

R_8 =>



Données à traiter :

Vecteurs piles *normalisés* (type : *float* ; taille : t).

Exemple :

```
Table = [    [20,30,10],
             [10,10,20],
             [30,0,10],
             [20,20,10]]
Vecteurs_norms = [  [0.3333,0.5,0.1667],
                    [0.25,0.25,0.5],
                    [0.75,0,0.25],
                    [0.4,0.4,0.2]]
```

.

Obtenir une égalité entre deux d'entre eux est difficile.

=>Utilisation de la Classification Hiérarchique Ascendante

Choix des distances pour l'apprentissage :

- barycentre d'une famille : Vecteur moyen de la famille. Noté : $b(U)$ avec U une famille.

- Distance entre vecteurs : Distance euclidienne. Notée : $d(x, y)$

- Distance entre familles : Distance de Ward entre deux familles. Notée : $D(U, V) = (\frac{|U||V|}{|U| + |V|})^{1/2} d(b(U), b(V))$

Critère d'arrêt :
Distance maximale de fusion des familles.
Notée M

=> On note $CHA(X0, R, M, n)$ le résultat obtenu pour l'étape n de table initiale $X0$.

Même ordre pour KS et CHA.
On compare $KS(R)$ et $CHA(X0, R, M, n)$.

- $KS(R) = CHA(X0, R, M, n)$ => équilibre atteint

- $KS(R) \neq CHA(X0, R, M, n)$

$$=> \left\{ \begin{array}{l} \text{L'équilibre n'est pas atteint} \\ \textit{ou} \\ \text{Des familles avec équilibre} \\ \text{trop proches ont fusionné } (*) \end{array} \right.$$

Problème :

Cas $(*)$: On ne trouve jamais d'équilibre.

Solution :

Faire varier les paramètres $(M,X0)$

On écarte le cas des familles trop proches.

On cherche le plus petit n tel que :
$\forall m \in [|n, +\infty[|, KS(R) = CHA(X0, R, M, m)$.
On le note $rg(X0, R, M)$

Cas du Pouilleux simplifié

Règle simplifiée : Déplacement du Valet de Pique

Nombre de joueurs : j (= nombre de piles)
Nombre de cartes : N (dont un Valet de Pique).

Chaque joueur reçoit N/j cartes, et on distribue le reste au hasard.

A chaque tour : Chaque joueurs tire une carte dans le jeu de son voisin de gauche.
=> C'est un graphe cyclique

On cherche à trouver le rang d'équilibre.
On a donc besoin seulement de 2 types de cartes : "Valet de Pique" et "Autre".

Paramètres :

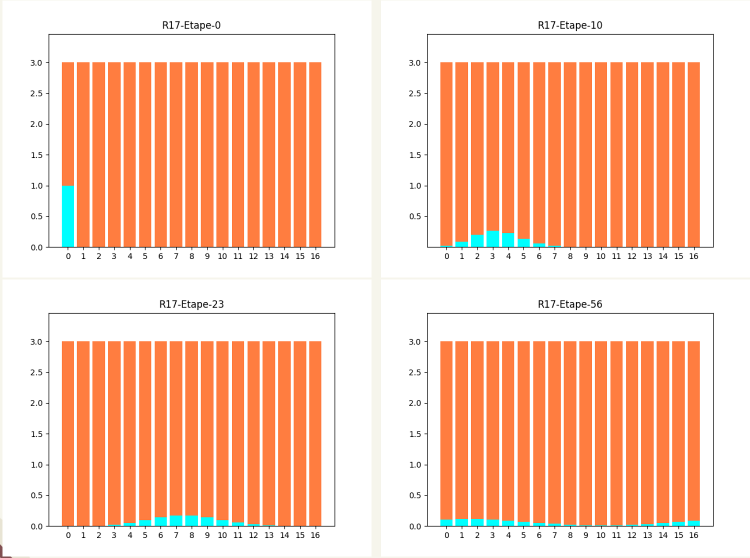
$$\begin{cases} j = 17 \\ N = 51 \text{ (cas classique)} \\ M = 0.01 \end{cases}$$

.

On obtient comme rang d'équilibre 175 tel que dans toutes les piles il y a :
- 1.8% de chances d'y piocher le Valet de Pique
- 98.2% de chances d'y piocher autre chose

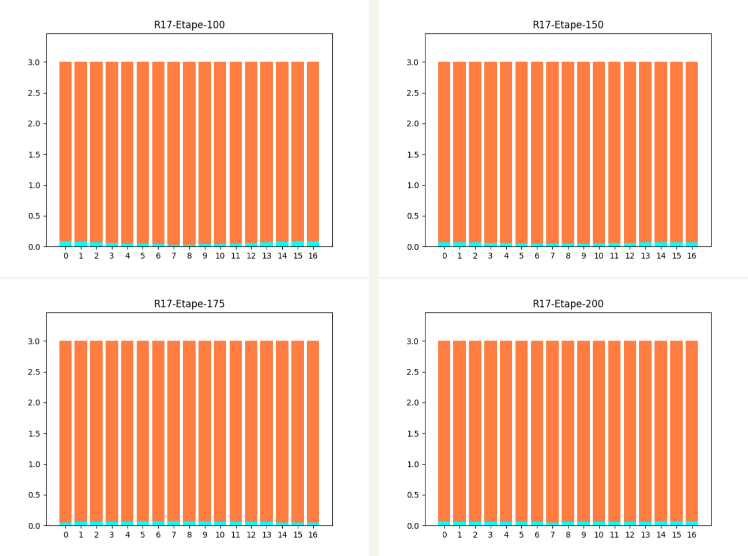
Cas du Pouilleux simplifié

Application



Cas du Pouilleux simplifié

Application



Bibliographie

1 MARVIN RAUSAND ET ARNLJOT HOYLAND : System Reliability Theory Model, Statistical Methods, and Applications (Chapitre 8 Markov Processus) : Second Edition, Wiley (2003), ISBN 9780471471332

2 PIERRE-LOIC MELIOT : Chaînes de Markov : théorie et applications : IMO université Paris-Saclay, https ://www.imo.universite-paris-saclay.fr/~pierre-loic.meliot/markov/markov.pdf (Date de téléchargement : 01/06/23)

3 MIGUEL ROTENBERG : Les Jeux de Société, essai sur la production d'un outil d'analyse autour des mécaniques de jeu, (Chapitre 6 : Mécaniques en lien avec le matériel du jeu) (6.1.4 La carte) : https ://www.ville-jeux.com/IMG/pdf/m2_-_mecaiques_des_jeux_de_societe.pdf (Date de téléchargement : 17/02/2023)

4 IMAGEIO CONTRIBUTORS : User guide : https ://imageio.readthedocs.io/en/stable /user_guide/index.html

5 NUMPY DEVELOPERS : Numpy.org : https ://numpy.org/doc/stable/reference/arrays. html

ANNEXES & OUTILS

Outils : Trie de couleurs contrastées

```
import matplotlib.pyplot as pyplot
'''
Programme permetant de choisir des couleurs
contrastées parmi une liste large.
'''
colors = [
(0.0, 1.0, 1.0),(1.0, 0.49, 0.251), (0.0, 0.0, 1.0), (0.498, 1.0, 0.0), (1.0, 1.0, 0.0),
(0.373, 0.62, 0.627), (0.647, 0.165, 0.165), (0.871, 0.722, 0.529), (0.816, 0.125, 0.565),
(0.863, 0.863, 0.863),(1.0, 0.843, 0.0), (0.251, 0.878, 0.816), (0.0, 0.78, 0.549),
(0.933, 0.51, 0.933), (0.98, 0.922, 0.843),(0.541, 0.169, 0.886),(0.612, 0.4, 0.122),
(1.0, 0.922, 0.804),  (0.929, 0.569, 0.129), (0.541, 0.212, 0.059), (0.541, 0.2, 0.141),
(1.0, 0.38, 0.012), (1.0, 0.6, 0.071),(0.824, 0.412, 0.118), (0.239, 0.349, 0.671),
(0.239, 0.569, 0.251), (0.502, 0.541, 0.529), (1.0, 0.498, 0.314), (0.392, 0.584, 0.929),
(0.89, 0.812, 0.341),(1.0, 0.894, 0.769), (0.863, 0.078, 0.235), (0.0, 0.933, 0.933),
(0.722, 0.525, 0.043), (0.0, 0.392, 0.0),(0.498, 1.0, 0.831),(0.741, 0.718, 0.42),
(0.333, 0.42, 0.184), (1.0, 0.549,0.0), (0.6, 0.196, 0.8), (0.914, 0.588, 0.478),
(0.561, 0.737, 0.561), (0.282, 0.239, 0.545), (0.184, 0.31, 0.31), (0.0, 0.808, 0.82),
...
(0.188, 0.502, 0.078), (0.329,1.0, 0.624), (1.0, 0.961, 0.933), (0.369, 0.149, 0.071),
(0.557, 0.22, 0.557), (0.773, 0.757, 0.667), (0.443, 0.776, 0.443), (0.49, 0.62, 0.753),
(0.557, 0.557, 0.22), (0.776, 0.443, 0.443), (0.443, 0.443, 0.776), (0.22, 0.557, 0.557),
(0.627, 0.322, 0.176), (0.753, 0.753, 0.753), (0.529, 0.808, 0.922),(0.416, 0.353,0.804),
(0.439, 0.502, 0.565), (1.0, 0.98, 0.98), (0.0, 1.0, 0.498), (0.275, 0.51, 0.706),
(0.824, 0.706, 0.549), (0.0, 0.502, 0.502), (0.847, 0.749, 0.847), (1.0, 0.388, 0.278)]

i=0 #Indice de départ
f=100 #Nombre de couleur à enlever (à partir de la fin)
pyplot.figure()
pyplot.grid()
pyplot.title('Differentes couleurs')
pyplot.scatter(range(len(colors)-i -f), [x for x in range(len(colors)-i -f)], s = 100,
                                                  color = colors[i:(len(colors)-f)])
pyplot.show()
```

Outils : Excel de création de matrice équilibrées (8x8)

0	1	0	4	10	5	2	0	22
1	0	5	2	0	9	5	0	22
1	2	0	0	10	1	2	6	22
0	7	0	3	0	0	2	10	22
1	5	0	9	2	2	0	3	22
10	4	7	0	0	0	1	0	22
8	1	4	4	0	5	0	0	22
1	2	6	0	0	0	10	3	22
22	22	22	22	22	22	22	22	

Somme en ligne

Somme en colonne

Taille max : 8

Matrice équilibrée ?

Oui

Modélisation

```python
import random as r
import matplotlib.pyplot as m

def echange_bilateral_kitaire (l1,l2,k):
    '''
    l1 : pile numéro 1
    l2 : pile numéro 2
    k : Nombre carte à échanger
    '''
    E12 = l1[0:k]
    E21 = l2[0:k]
    l1 = E21 + l1[k:len(l1)]
    l2 = E12 + l2[k:len(l2)]
    return [l1,l2]

def melange_pile (l):
    '''
    l : pile à mélanger
    '''
    r.shuffle(l)

def affichage (table):
    '''
    table : table à afficher
    '''
    ligne = ""
    for i in table:
        for j in i:
            if j:
                ligne = ligne + ""
            elif not j:
                ligne = ligne +""
        ligne = ligne + "  "
    print(ligne)
```

```python
def melange_table (table):
    '''
    table : table à mélanger
    '''
    for i in range(0, len(table)):
        melange_pile(table[i])

def initialise_table (n_min,n_max,n_pile):
    '''
    n_min : Nombre minimum de cartes dans une pile
    n_max : Nombre maximum de cartes dans une pile
    n_pile : nombre de piles
    '''
    table = []
    for p in range(0,n_pile):
        longueur = r.randint(n_min, n_max)
        cibles = r.randint(0,longueur)
        pile =[]
        if cibles != 0:
            for j in range(0,cibles):
                pile.append(True)
        if cibles != longueur:
            for j in range(cibles,longueur):
                pile.append(False)
        table.append(pile)
    melange_table(table)
    print("Table base   " ,end='')
    affichage(table)
    return table
```

Modélisation

```python
def initialise_table_2 (n_min,n_max,n_pile,n_carte):
    '''
    n_min : Nombre minimum de cartes dans une pile
    n_max : Nombre maximum de cartes dans une pile
    n_pile : Nombre de piles
    n_carte : Nombre total de cartes (> n_min * n_pile)
    '''
    table = []
    for p in range(0,n_pile -1) :
        longueur = r.randint(n_min, min(n_max,n_carte - (n_pile -1 -p)))
        n_carte = n_carte - longueur
        cibles = r.randint(0,longueur)
        pile =[]
        if cibles != 0:
            for j in range(0,cibles):
                pile.append(True)
        if cibles != longueur:
            for j in range(cibles,longueur):
                pile.append(False)
        table.append(pile)
    longueur = n_carte
    cibles = r.randint(0,longueur)
    pile =[]
    if cibles != 0:
        for j in range(0,cibles):
            pile.append(True)
    if cibles != longueur:
        for j in range(cibles,longueur):
            pile.append(False)
    table.append(pile)
    melange_table(table)
    print("Table base   " ,end='')
    affichage(table)
    return table
```

```python
def compte_pile (l):
    '''
    l : pile à dénombrer
    '''
    compteur =0
    for i in range(0,len(l)):
        if l[i] :
            compteur =compteur + 1
    return compteur

def compte_table (table):
    '''
    table : table à dénombrer
    '''
    compteur = []
    for i in range(0,len(table)):
        compteur.append(compte_pile(table[i]))
    return compteur
```

Modélisation

```python
def experience_2_piles(n_echange,n_min,n_max,taille_paquet) :
    '''
    n_echange : nombre d'echange
    n_min : Nombre minimum de cartes dans une pile
    n_max : Nombre maximum de cartes dans une pile
    taille_paquet : Nombre de cartes échangées par échange
    '''
    table = initialise_table (n_min,n_max,2)
    compteur = compte_table(table)
    repartition_tot = (compteur[0] + compteur[1])/(len(table[0]) + len(table[1]))
    #Data graphique
    x = [0]
    y1 = [(compteur[0]/len(table[0])) /repartition_tot]
    y2 = [(compteur[1]/len(table[1])) /repartition_tot]
    #Exécution des échanges
    for i in range(0,n_echange):
        table = echange_bilateral_kitaire(table[0],table[1],taille_paquet)
        compteur = compte_table(table)
        melange_table(table)
        x.append(i+1)
        y1.append((compteur[0]/len(table[0])) /repartition_tot)
        y2.append((compteur[1]/len(table[1])) /repartition_tot)
        print("Echange n°",i+1,"  ",y1[i],"/",y2[i],"   ",end='')
    #Graphique
    m.bar(x,y1,alpha=0.25)
    m.bar(x,y2,alpha=0.25)
    m.hlines([1],[-0.66],[n_echange+0.66])
    m.xlabel("Echange n°")
    m.ylabel("Proportion de la pile / Proportion totale")
    m.title("Répartition des piles")
    m.text(0,0.05,"Taille pile n°1 : "+str(len(table[0]))
           + " \nTaille pile n°2 : " +str(len(table[1]))
           + " \n Proportion globale : "+ str(repartition_tot))
    m.show()
```

Modélisation

```python
def echange_circulaire_kitaire (table,k):
    '''
    table : Liste des piles à échanger
    k : Nombre de cartes à échanger
    '''
    paquets = []

    for i in range(0,len(table)):
        paquets.append(table[i][0:k])
    for i in range(0,len(table)):
        table[i] = paquets[i-1] + table[i][k:len(table[i])]

    return table

def moyenne (l) :
    moy = []
    for j in range (0,len(l[0])):
        s=0
        for i in range(0,len(l)):
            s = s + l[i][j]
        moy.append(s/len(l))
    return moy
```

Modélisation

```
def experience_n_piles(n,n_echange,n_min,n_max,taille_paquet):'''
    n : Nombre de piles
    n_echange : nombre d'echange
    n_min : Nombre minimum de cartes dans une pile
    n_max : Nombre maximum de cartes dans une pile
    taille_paquet : Nombre de cartes échangées par échange'''
    table = initialise_table (n_min,n_max,n)
    compteur = compte_table(table)
    compteur_tot, cartes_tot = 0,0
    for i in range(0,n):
        compteur_tot += compteur[i]
        cartes_tot += len(table[i])
    repartition_tot = compteur_tot/cartes_tot
    #Data graphique
    x,y,moy = [0],[],[]
    for i in range(0,n):
        y.append([(compteur[i]/len(table[i])) /repartition_tot])
    moy.append(moyenne(y))
    #Execution
    for i in range(0,n_echange) :
        table = echange_circulaire_kitaire(table,taille_paquet)
        compteur = compte_table(table)
        melange_table(table)
        x.append(i+1)
        for j in range(0,n):
            y[j].append((compteur[j]/len(table[j])) /repartition_tot)
        moy.append(moyenne(y))
    for i in range(0,n):
        m.bar(x,y[i],alpha=0.25)
    m.plot(x,moy)
    m.xlabel("Echange n°")
    m.ylabel("Proportion de la pile / Proportion totale")
    m.title("Répartition des piles")
    m.show()
```

Modélisation

```python
def echange_matriciel(table,matrice):
    '''
    table : Liste de piles à échanger
    matrice : matrice de relation entre les piles
    Exemple :
    [[1,2,3,0,2,0],
     [0,0,0,0,1,0],
     [1,2,3,0,0,3],
     [0,0,4,0,1,0],
     [0,1,0,0,0,0],
     [0,0,0,0,1,2]]
    '''

    paquets = []

    for i in range(0,len(matrice)):
        paquets.append([])
        for j in range(0,len(matrice)) :
            k = matrice[i][j]
            p=table[i][0:k]
            paquets[i].append(p)
            table[i] = table[i][k:len(table[i])]
    #print (table)
    #print(paquets)
    for i in range(0,len(matrice)):
        for j in range(0,len(matrice)) :
            table[j] = table[j]+paquets[i][j]

    #print (table)
    return table
```

```python
def verif_equilibre_matrice (matrice):
    '''
    Permet de vérifier si une matrice est équilibrée
    ------------------------------------------------
    matrice : matrice de relation entre les piles
    '''

    res = True
    for i in range(0,len(matrice)):
        si=0
        sj=0
        for j in range (0,len(matrice)):
            si = si+ matrice[i][j]
            sj = sj+ matrice[j][i]
        if sj != si :
            res= False
    return res
```

Modélisation

```
def experience_matricielle(n,n_echange,n_min,n_max,matrice):'''
    n : Nombre de piles
    n_echange : nombre d'échanges
    n_min : Nombre minimum de cartes dans une pile
    n_max : Nombre maximum de cartes dans une pile
    matrice : matrice de relation entre les piles'''
    table = initialise_table (n_min,n_max,n)
    compteur = compte_table(table)
    compteur_tot,cartes_tot = 0,0
    for i in range(0,n):
        compteur_tot += compteur[i]
        cartes_tot += len(table[i])
    repartition_tot = compteur_tot/cartes_tot
    legende = []
    for i in range(0,n):
        legende.append ("Pile n°"+str(i)+" : "
            +str(compteur[i])+"/"+str(len(table[i])))
    #Data graphique
    x,y,moy = [0],[],[]
    for i in range(0,n):
        y.append([(compteur[i]/len(table[i])) /repartition_tot])
    moy.append(moyenne(y))
    #Execution
    for i in range(0,n_echange) :
        table = echange_matriciel(table,matrice)
        compteur = compte_table(table)
        melange_table(table)
        x.append(i+1)
        for j in range(0,n):
            if len(table[j]) != 0 :
                y[j].append((compteur[j]/len(table[j])) /repartition_tot)
            else:
                y[j].append(0)
```

```
moy.append(moyenne(y))
for i in range(0,n):
    legende[i] = legende[i] + " -> Pile n°"+str(i)
    +" : "+str(compteur[i])+"/"+str(len(table[i]))
    m.bar(x,y[i],alpha=0.25)
m.legend(legende)
m.plot(x,moy)
m.xlabel("Echange n°")
m.ylabel("Proportion de la pile/Proportion totale")
m.title("Répartition des piles")
m.show()
```

Modélisation

```
def experience_matricielle_2 (table,n_pile,n_echange,matrice,show):'''
    table : Table de départ
    n_pile : Nombre de piles
    n_echange : nombre d'échanges
    n_min : Nombre minimum de cartes dans une pile
    n_max : Nombre maximum de cartes dans une pile
    n_carte : Nombre totale de cartes (> n_min * n_pile)
    matrice : matrice de relation entre les piles'''
    compteur = compte_table(table)
    compteur_tot,cartes_tot = 0,0
    for i in range(0,n_pile):
        compteur_tot += compteur[i]
        cartes_tot += len(table[i])
    repartition_tot = compteur_tot/cartes_tot
    legende = []
    for i in range(0,n_pile):
        legende.append ("Pile n°"+str(i)+" : "+str(compteur[i])+"/"+str(len(table[i])))
    #Data graphique
    x,y,moy = [0],[],[]
    for i in range(0,n_pile):
        y.append([(compteur[i]/len(table[i])) /repartition_tot])
    #Execution
    for i in range(0,n_echange) :
        table = echange_matriciel(table,matrice)
        compteur = compte_table(table)
        melange_table(table)
        x.append(i+1)
        for j in range(0,n_pile):
            if len(table[j]) != 0 :
                y[j].append((compteur[j]/len(table[j])) /repartition_tot)
            else:
                y[j].append(0)
    moy=moyenne(y)
```

Modélisation

```
if show :
    for i in range(0,n_pile):
        legende[i] = legende[i] + " -> Pile n°"+str(i)+" : "+str(compteur[i])+"/"+str(len(table[i]))
        m.bar(x,y[i],alpha=0.25)
    m.legend(legende)
    m.plot(x,moy,label="Moyenne à l'étape")
    m.hlines([1],[-0.66],[n_echange+0.66],label='Equilibre global')
    m.xlabel("Echange n°")
    m.ylabel("Proportion de la pile / Proportion totale")
    m.title("Répartition des piles")
    m.axis(xmin=0, ymin=0)
    m.show()
return moy
```

Modélisation

```
def etude_experiences_matricielles(table,matrice,n_pile,n_echange,n_test):
    '''
    Permet de représenter les moyennes de répartition des piles à
    chaque étape pour une série d'expérience.
    Permet de visualiser une moyenne des moyennes.
    ------------------------------------------------
    table : Table de départ
    matrice : matrice de relation entre les piles
    n_pile : Nombre de piles
    n_echange : nombre d'échanges
    n_test : Nombre d'experiences successives
    '''
    etape = []
    for j in range(0,n_echange+1):
        etape.append(j)

    moy=[]
    for k in range(0,n_test):
        moy.append(experience_matricielle_2(table,n_pile,n_echange,matrice,False))

    moymoy = moyenne(moy)
    m.plot(etape,moymoy,linewidth=1,color='b')
    for i in range(0,len(moy)):
        m.plot(etape,moy[i],linewidth=0.2)

    m.legend(["Moyenne des moyennes"])
    m.xlabel("Etape n°")
    m.ylabel("Moyennes des experiences")

    m.axis(xmin=0, ymin=0)
    m.show()
```

Simulation (2-types)

```python
import random as r
import matplotlib.pyplot as m
import numpy as np

def conversion (table) : '''
    Transforme une table réelle
    en table de quantité
    ------------------------------
    table : Table réelle'''
    n = len(table)
    l=[]
    for i in range (0,n):
        m = len(table[i])
        c = 0
        for j in range(0,m):
            if table[i][j] :
                c += 1
        l.append([(m-c) + c*1j])
    T1 = np.array(l)
    return T1

def proba (M) :   '''
    Transforme une table de
    quantité en table de proportion
    ---------------------------------
    matrice : matrice de relation
    entre les piles'''
    I = M.imag
    R = M.real
    P = I/(I+R)
    return P
```

```python
def next_etape (T1,R,C) :
    '''
    Passage de récurence de la table des cartes.
    ------------------------------------------------
    T1 : matrice des cartes à l'étape courante
    R : matrice de relations
    C : matrice colonne
    '''
    P = proba(T1)
    T2 = T1 - np.dot(R,C)*((-1+ 1j)*P+C) + np.dot(R.T,((-1+ 1j)*P+C))

    return T2

def verif_equilibre_matrice (matrice):
    '''
    Permet de vérifier si une matrice est équilibrée
    ------------------------------------------------
    matrice : matrice de relation entre les piles
    '''

    res = True
    for i in range(0,len(matrice)):
        si=0
        sj=0
        for j in range (0,len(matrice)):
            si = si+ matrice[i][j]
            sj = sj+ matrice[j][i]
        if sj != si :
            res= False
    return res
```

Simulation (2-types)

```
def execution_echange(T1,R,n,e):
    '''
    Echange les cartes, calcul la moyenne et affiche.
    ------------------------------------------------
    T1 : Table initiale (réelle)
    R : Matrice de relation entre les piles
    n : Nombre de piles
    e : Nombre d'étapes
    '''


    l =[]
    for i in range(0,n):
        l.append([1])
    C= np.array(l)
    etape = [0]

    Prop = proba(T1)
    proportion_totale = (proba(np.dot((T1.T),C)))[0][0]
    print("proportion_totale : ",proportion_totale)

    liste_proportions_reduites = (Prop/proportion_totale).tolist()

    T2=T1

    for j in range(e):
        T2 = next_etape(T2,R,C)
        Prop2 = (((proba(T2)/proportion_totale).T)[0]).tolist()
        etape.append(j+1)
        for i in range(0,len(liste_proportions_reduites)):
            liste_proportions_reduites[i].append(Prop2[i])
```

Simulation (2-types)

```
moy = []
for e in range (0,len(liste_proportions_reduites[0])):
    temp = 0
    for k in range (0,len(liste_proportions_reduites)):
        temp += liste_proportions_reduites[k][e]
    moy.append(temp)

moy = np.array(moy)
moy = moy/len(liste_proportions_reduites)


for j in range (0,len(liste_proportions_reduites)):
    m.bar(etape,liste_proportions_reduites[j],alpha=0.25)


m.xlabel("Echange n°")
m.ylabel("Proportion de la pile / Proportion totale")
m.title("Répartition des piles")

legende = []
for i in range(0,n):
    legende.append ("Pile n°"+str(i))
m.legend(legende)
m.plot(etape,moy,label="Moyenne à l'étape")

m.show()
```

Simulation (Gif)

```python
import random as r
import matplotlib.pyplot as m
import numpy as np
import imageio.v2 as im
import os
from pathlib import Path
import copy
```

```python
path = os.path.abspath(Path(__file__).parent)
def table_alea (n,N,t,b) :'''
    Permet de créer une table aléatoirement suivant certains paramètres
    ------------------------------------------------
    n       : nombre de piles
    N       : nombre de cartes
    t       : nombre de types de cartes différent
    b       : taille minimum de pile    '''
    quantitable,proportable = [],[]

    #QUANTITABLE
    N = N - b*n
    borne =sorted(r.sample(range(0,N),n-1))
    borne.append(N)
    temp = 0
    for j in range(0,n) :
        quantitable.append(borne[j] - temp +b)
        temp = borne[j]

    #PROPORTABLE
    for i in range(0,n):
        pile = []
        range_q = r.randrange(0,quantitable[i]+1)
        to_be_sorted = [range_q for k in range(0,t-1)]
        pourcent = sorted(to_be_sorted)
        pourcent.append(quantitable[i])
        temp=0
        for j in range(0,t) :
            pile.append((pourcent[j] - temp)/quantitable[i])
            temp = pourcent[j]
        proportable.append(pile)
    Q,P=(np.array(quantitable)).reshape(n,1),np.array(proportable)
    return Q,P
```

Simulation (Gif)

```python
def simulation (n,N,t,b,e,R): '''
    n       : nombre de piles
    N       : nombre de cartes
    t       : nombre de types de cartes différent
    b       : taille minimum de pile
    e       : nombre d'étapes
    R       : matrice de relation '''
    Q,P = table_alea(n,N,t,b)
    historique = [[copy.deepcopy(Q),copy.deepcopy(P)]]
    C = np.ones((n,1))
    L = np.ones((1,t))
    for k in range(0,e) :
        P_=copy.deepcopy(P)
        Q_=copy.deepcopy(Q)
        S = np.zeros((n,t))
        P_2 = np.zeros((n,t))

        for i in range(0,n):
            for j in range(0,t):
                s=0
                for k in range(0,n):
                    s+= R[k][i]*P[k][j]
                S[i][j]+=s

        Q= copy.deepcopy(Q_ - (R @ C)+ (R.T @ C))

        for i in range(0,n):
            for j in range(0,t):
                P_2[i][j] += 1/(Q[i])
        P= copy.deepcopy(P_2*(P_ * ((Q_ - (R @ C))@ L) + S))

        historique.append([copy.deepcopy(Q),copy.deepcopy(P)])
    return historique
```

Simulation (Gif)

```
def afficher_histo (n,t,etat,name,save,show,e):'''
    n       : nombre de piles
    t       : nombre de types de cartes différent
    état    : état de la table au départ
    save    : Sauvegarder ou non l'histogramme
    show    : Montrer ou non l'histogramme
    e       : nombre d'étapes     '''
    P = copy.deepcopy(etat[1])
    Q = copy.deepcopy(etat[0])
    max = np.max(Q)
    c = colors
    x=(np.linspace(1,n,n))

    L = np.ones((1,t))
    E= copy.deepcopy((P*(Q@L)).T)

    #Fond pour la stabilité
    m.bar(x,[max for x in P],color='white',alpha=0.0)
    m.figure(facecolor='#F6F5EC')
    B = np.zeros((1,n))
    for j in range(0,t):
        #Types suivants
        m.bar(x,E[j],bottom=B[0],color=c[j])
        B = copy.deepcopy(B + E[j])

    m.title("Etape "+str(e))
    m.xticks(x,x)

    if save :
        m.savefig(path+'/Gif/'+name)
    if show:
        m.show()
    m.clf()
```

Simulation (Gif)

```
def build_gif (n,t,historique) :
    '''
    n           : nombre de piles
    t           : nombre de types de cartes différent
    historique  : List des états de la table pour chaque étape
    '''
    Lim = []
    for j in range(0,len(historique)):
        print('gif'+str(j))
        afficher_histo(n,t,historique[j],'gif'+str(j),True,False,j)
        Lim.append('gif'+str(j))

    with im.get_writer(path+'/Gif/'+'mygif.gif',mode='I',fps=2) as writer :
        for filename in Lim :
            image = im.imread(path+'/Gif/'+filename + '.png')
            writer.append_data(image)
```

Simulation (Markov) - Outils calculs

```
def prod_mat(A,B):
    ''' A : matrice carrée
        B : matrice carrée '''
    C = []
    for i in range(0,len(A)):
        Cl=[]
        for j in range(0,len(A)):
            s = 0
            for k in range(0,len(A)):
                s += A[i][k]*B[k][j]
            Cl.append(s)
        C.append(Cl)
    return C

def scal_prod_mat(a,A):
    ''' a : scalaire
        A : Matrice'''
    C=A
    for i  in range(0,len(A)):
        for j in range(0,len(A[0])):
            C[i][j]= a*A[i][j]
    return C

def scal_div_mat(a,A):
    ''' a : scalaire
        A : Matrice'''
    C=A
    for i  in range(0,len(A)):
        for j in range(0,len(A)):
            C[i][j]= A[i][j]/a
    return C
```

```
def pwr_mat(Q,n):
    ''' n : Puissance
        Q : Matrice'''
    C = copy.deepcopy(Q)
    for i in range(0,n-1):
        C = prod_mat(C,Q)
    return C

def dif_mat(A,B):
    ''' A : matrice
        B : matrice  '''
    C=[]
    for i in range(0,len(A)):
        Cl=[]
        for j in range(0,len(A[0])):
            Cl.append(A[i][j]-B[i][j])
        C.append(Cl)
    return C

def dif_list(A,B):
    ''' A : Liste
        B : Liste '''
    C=[]
    for i in range(0,len(A)):
        C.append(A[i]-B[i])
    return C
def abs_list(A):
    ''' A : Liste'''
    C=[]
    for i in range(0,len(A)):
        C.append(abs(A[i]))
    return C
```

Simulation (Markov) - Outils calculs

```
def sum_list(A,B):
    ''' A : Liste
        B : Liste '''
    C = []
    for i in range(len(A)):
        C.append(A[i]+B[i])
    return C

def scal_prod_list(a,L):
    ''' a : scalaire
        L : Liste'''
    C=[]
    for i in range(0,len(L)):
        C.append(a*L[i])
    return C

def scal_div_list(a,L):
    ''' a : scalaire
        L : Liste'''
    C=[]
    for i in range(0,len(L)):
        l = L[i]
        C.append(l/a)
    return C
```

```
def sum_pile(X):
    '''Renvoie la liste de la somme
    des coordonnées de chaque vecteur
    ---------------------
    X : Liste de vecteurs'''
    N= []
    for i in range(0,len(X)):
        s=0
        for j in range(0,len(X[0])):
            s += X[i][j]
        N.append(s)
    return N

def max_pile(X):
    '''Renvoie le maximum de
    la liste de la somme des
    coordonnées de chaque vecteur
    -----------------------
    X : Liste de vecteurs'''
    N=sum_pile(X)
    m=0
    for i in range(0,len(N)):
        if N[i]>m :
            m = N[i]
    return m
```

Simulation (Markov) - Outils calculs

```python
def transpose(X):
    '''X : Matrice'''
    p = len(X)
    t = len(X[0])
    C =[]
    for j in range(0,t):
        Cl = []
        for i in range(0,p):
            Cl.append(X[i][j])
        C.append(Cl)
    return C

def piles_repartition(X):
    '''Divise les vecteur par
    la somme de leur coordonnées
    --------------------------
    X : Liste de vecteurs '''
    N=sum_pile(X)
    Y=[]
    p = len(X)
    t = len(X[0])

    for i in range(0,p):
        Yl = []
        for j in range(0,t):
            Yl.append(X[i][j]/N[i])
        Y.append(Yl)
    return Y
```

```python
def sum_coef_list(L):
    '''Somme de la liste
    -------------------
    L :  Liste'''
    s=0
    for i in range(0,len(L)):
        s += L[i]
    return s

def min_list(L):
    '''L : liste'''
    min = L[0]
    for e in L:
        if e<min:
            min = e
    return min
```

Simulation (Markov) - Conversion

```
def transforme_R_Q (R,X0,p,t):
    ''' Transforme R en la matrice de Markov Q*Nmax (en partant de X0)
    et une constante de division Nmax
    ---------------------------------
    R   : Matrice d'adjacence
    X0  : Table initiale
    p   : Nombre de pile
    t   : Nombre de type'''
    N = []
    for i in range(0,p):
        s=0
        for j in range(0,t):
            s+=X0[i][j]
        N.append(s)
    Nmax = 1
    for i in range(0,p):
        Nmax = Nmax*N[i]
    Q=[]
    for i in range(0,p):
        Ql=[]
        for j in range(0,p):
            if i==j :
                s = 0
                for k in range(0,p):
                    if k!=i:
                        s+= R[j][k]
                l = int(int(Nmax)-int(s*(Nmax/N[i])))
                Ql.append(l)
            else :
                Ql.append(int((R[i][j])*int(Nmax/N[i])))
        Q.append(Ql)
    return Q,Nmax
```

Simulation (Markov) - Conversion

```
def etape(X,Q,n,Nmax):
    ''' Renvoie Xn
    ------------------------
    X   : Table
    Q   : Matrice de Markov
    n   : Nombre d'étapes
    Nmax: Constante de division
    '''
    Qn = scal_div_mat(int(Nmax**n),pwr_mat(Q,n))
    Xn=[]
    for j in range(0,len(X)):
        s = scal_prod_list(Qn[0][j],X[0])
        for i in range(1, len(X)):
            s = sum_list(s,scal_prod_list(Qn[i][j],X[i]))
        Xn.append(s)
    return Xn

def etape_succ(X,Q,n,Nmax):
    ''' Renvoie la liste [X,...,Xn]
    ------------------------
    X   : Table
    Q   : Matrice de Markov
    n   : Nombre d'étapes
    Nmax: Constante de division
    '''
    H=[X]
    for e in range (1,n+1):
        H.append(etape(X,Q,e,Nmax))
    return H
```

Simulation (Markov) - Affichage

```
def mono_image(nom,X,affiche):
    ''' Enregistre le graphique de l'état de la table X et renvoie son adresse
    -----------------------------------------------------
    nom     : Nom de l'image
    X       : état de la table
    affiche : Spécifie si le graphique doit s'afficher'''
    Y = transpose(X)
    pile = []
    maxX=[]
    base = []

    max = 1.1*max_pile(X)
    p = len(X)
    t = len(X[0])

    for i in range(0,p):
        pile.append(i)
        maxX.append(max)
        base.append(0)

    m.bar(pile,maxX,color='black',alpha=0.0)

    for j in range(0,t):
        m.title(nom)
        m.bar(pile,Y[j],bottom=base,color=colors[j],alpha=1.0)
        base = sum_list(Y[j],base)

    m.xticks(pile,pile)
    m.savefig(path+'/Gif/'+nom+".png")
    if affiche :
        m.show()
    m.clf()
    return path+'/Gif/'+nom+".png"
```

Simulation (Markov) - Affichage

```
def gif (nom,X,Q,n,affiche,Nmax):
    H = etape_succ(X,Q,n,Nmax)
    i = 0
    Lim=[]
    for Xn in H :
        Lim.append(mono_image(nom+str(i),Xn,affiche))
        i+=1

    with im.get_writer(path+'/Gif/'+nom+'.gif',mode='I',fps=5) as writer :
        for filename in Lim :
            image = im.imread(filename)
            writer.append_data(image)
```

Simulation (Markov) - Affichage

```
def gif (nom,X,Q,n,affiche,Nmax):
    H = etape_succ(X,Q,n,Nmax)
    i = 0
    Lim=[]
    for Xn in H :
        Lim.append(mono_image(nom+str(i),Xn,affiche))
        i+=1

    with im.get_writer(path+'/Gif/'+nom+'.gif',mode='I',fps=5) as writer :
        for filename in Lim :
            image = im.imread(filename)
            writer.append_data(image)
```

Simulation (Markov) - Apprentissage

```
def fusionne_fam(X,i,j):
    ''' Fusionne les familles i et j
    -------------------------------
    X        : Liste de familles
    i et j   : Indices des familles
    '''
    X0 =[]
    for k in range(0,len(X)):
        if k == i :
            X0.append(   [X[i][0] + X[j][0]]+ X[i][1:(len(X[i]))] + X[j][1:(len(X[j]))]    )
        elif k != j :
            X0.append(X[k])
    print("Fus : ",i,j)
    print(X)
    print(X0)
    return X0

def distance_vec (x,y) :
    '''
    x et y : vecteurs
    '''
    d = abs_list(dif_list(x,y))
    s=0
    for i in range(0,len(x)):
        s += d[i]*d[i]
    return np.sqrt(s)

def barycentre_fam(U):
    '''U : Famille'''
    s= copy.deepcopy(U[1])
    for i in range(2,len(U)):
        s = sum_list(s,U[i])
    return scal_div_list((len(U)-1),s)
```

Simulation (Markov) - Apprentissage

```
def distance_fam_Ward (U,V):
    ''' U et V : Familles'''
    bu = barycentre_fam(U)
    bv = barycentre_fam(V)
    return (np.sqrt(((len(U)-1)*(len(V)-1))/(len(U)+len(V)-2)))*distance_vec(bu,bv)

def classification_hierarchique_ascendante(X,M):
    ''' X   : Etat de la table
        M   : Condition d'arrêt (Distance maximum de fusion)'''
    X0=[]

    for k in range(0,len(X)):
        X0.append([[k],tuple(scal_div_list(sum_coef_list(X[k]),X[k]))])

    print(X0)
    d_f_min=0
    while len(X0)>1 and d_f_min<M:
        d_f_min = distance_fam_Ward(X0[0],X0[1])
        pair_min = (0,1)
        for i in range(0,len(X0)):
            for j in range(i+1,len(X0)):
                if distance_fam_Ward(X0[i],X0[j]) < d_f_min:
                    d_f_min = distance_fam_Ward(X0[i],X0[j])
                    pair_min = (i,j)
        if d_f_min<M :
            X0 = fusionne_fam(X0,pair_min[0],pair_min[1])

    return X0
```

Simulation (Markov) - Affichage Graphe

```python
def graphe_fam(F):
    '''
    Enregistre le graphe de la table en regroupant les piles par famille (cluster)
    ------------------------------------------------
    F : Famille
    '''
    dot = graphviz.Digraph('Table',format='png')
    for k in range(0,len(F)):
        c = graphviz.Digraph(name="cluster_"+str(k))
        c.attr(label="Famille "+str(k))
        for l in F[k]:
            c.node(str(l))
        dot.subgraph(c)
    dot.render(directory='doctest-output', view=True)
```

Simulation (Markov) - Composantes connexes

```python
def parcours_profondeur_postordre (R):
    etat =[]
    parent=[]
    debut=[]
    fin=[]
    postordre =[]
    for i in range(0,len(R)):
        etat.append(0)
        parent.append(i)
        debut.append(-1)
        fin.append(-1)
    temps=0
    def visiter (R,k,etat,parent,debut,fin,postordre,temps):
        (etat2,parent2,debut2,fin2,postordre2,temps2) = (etat,parent,debut,fin,postordre,temps+1)
        debut2[k]=temps2
        etat2[k]=1
        for j in range(0,len(R)):
            if R[k][j]>0 and etat2[j]==0:
                etat2[j]=k
                (etat2,parent2,debut2,fin2,postordre2,temps2)=
                            visiter(R,j,etat2,parent,debut,fin,postordre,temps+1)
        temps2 +=1
        fin2[k]=temps2
        postordre2.append(k)
        etat2[k]=2
        return (etat2,parent2,debut2,fin2,postordre2,temps2)

    for i in range(0,len(R)):
        if etat[i]==0 :
            (etat,parent,debut,fin,postordre,temps)=visiter(R,i,etat,parent,debut,fin,postordre,temps)
    return postordre
```

```
'''
Renvoie la liste de post-parcours
en profondeur de G(R)
-----------------------------------
R   : Matrice d'adjacence
'''
```

Simulation (Markov) - Composantes connexes

```
def parcours_profondeur_cpst_frtmt_cnx(R,L):
    '''
    Renvoie la liste des composantes
    fortement connexes de G(R)
    -----------------------------------
    R   : Matrice d'adjacence transposée
    L   : Liste post-ordre de parcours
          en profondeur renversée
    '''
    etat =[]
    composantes_fortement_connexes =[]
    f=-1
    for i in range(0,len(R)):
        etat.append(0)
    temps=0

    def visiter (R,k,etat,composantes_fortement_connexes,f,temps):

        (etat2,composantes_fortement_connexes2,f2,temps2)=(etat,composantes_fortement_connexes,f,temps)
        temps2 +=1
        etat2[k]=1
        for j in range(0,len(R)):
            if R[k][j]>0 and etat2[j]==0:
                etat2[j]=k
                (etat2,composantes_fortement_connexes2,f2,temps2) =
                        visiter(R,j,etat2,composantes_fortement_connexes2,f2,temps2)
        temps2 +=1
        etat2[k]=2
        composantes_fortement_connexes2[f2].append(k)
        return (etat2,composantes_fortement_connexes2,f2,temps2)
    print(L)
    for s in L:
        if etat[s]==0 :
            f+=1
            composantes_fortement_connexes.append([])
            (etat,composantes_fortement_connexes,f,temps)=
                    visiter(R,s,etat,composantes_fortement_connexes,f,temps)
    return composantes_fortement_connexes
```

Simulation (Markov) - Famille

```
def kosaraju(R):
    ''' R   : Matrice d'adjacence     '''
    postordre_parcours = parcours_profondeur_postordre(R)
    print(postordre_parcours)
    composantes_fortements_connexes =
            parcours_profondeur_cpst_frtmt_cnx(transpose(R),list(reversed(postordre_parcours)))
    return composantes_fortements_connexes
```

```
def transforme_FamVect_Fam(X0):
    ''' Retire les vecteur et ne
    garde que les indices de familles
    --------------------------------
    X0 : Famille de vecteur'''
    F = []
    for i in range(0,len(X0)):
        F.append(X0[i][0])
    return F
```

```
def trie_Famille(F):
    ''' Trie une famille selon au ordre arbitraire'''
    F_min=[]
    for k in range(0,len(F)):
        F_min.append(min_list(F[k]))
    max =0
    for i in range(0,len(F_min)):
            if F_min[i]>max:
                max = F_min[i]
    list_min=[]
    for j in range(0,len(F_min)):
        min=F_min[0]
        imin=0
        for i in range(0,len(F_min)):
            if F_min[i]<min:
                imin=i
                min = F_min[i]
        list_min.append(imin)
        F_min[imin]=max+1
    Fam=[]
    for k in range(0,len(list_min)):
        Fam.append(sorted(F[list_min[k]]))
    return(Fam)
```

Simulation (Markov) - équilibre

```python
def test_equilibre(R,X,p,t,n,M):
    '''Compare CHA et K à toutes les étapes
    ------------------------
    R   : Matrice d'adjacence
    X   : Table de départ
    p   : Nombre de pile
    t   : Nombre de type
    n   : Nombre d'étape à tester
    M   : Critère d'arrêt '''
    equilibre =[]
    K=trie_Famille(kosaraju(R))
    Q,Nmax=transforme_R_Q(R,X,p,t)
    for i in range(1,n+1):
        Xi=etape(X,Q,i,Nmax)
        Fi=trie_Famille(transforme_FamVect_Fam(classification_hierarchique_ascendante(Xi,M)))
        equilibre.append(K==Fi)
    return equilibre

def rang_equilibre(T):
    '''Renvoie l'indice du
    premier "True" dans T
    ----------------------
    T   : Liste de booléen'''
    for i in range(0,len(T)):
        if T[i]:
            return (True,i)
    return (False,len(T))
```

FIN