

# Le dilemme du prisonnier

# **Le Dilemme du prisonnier**

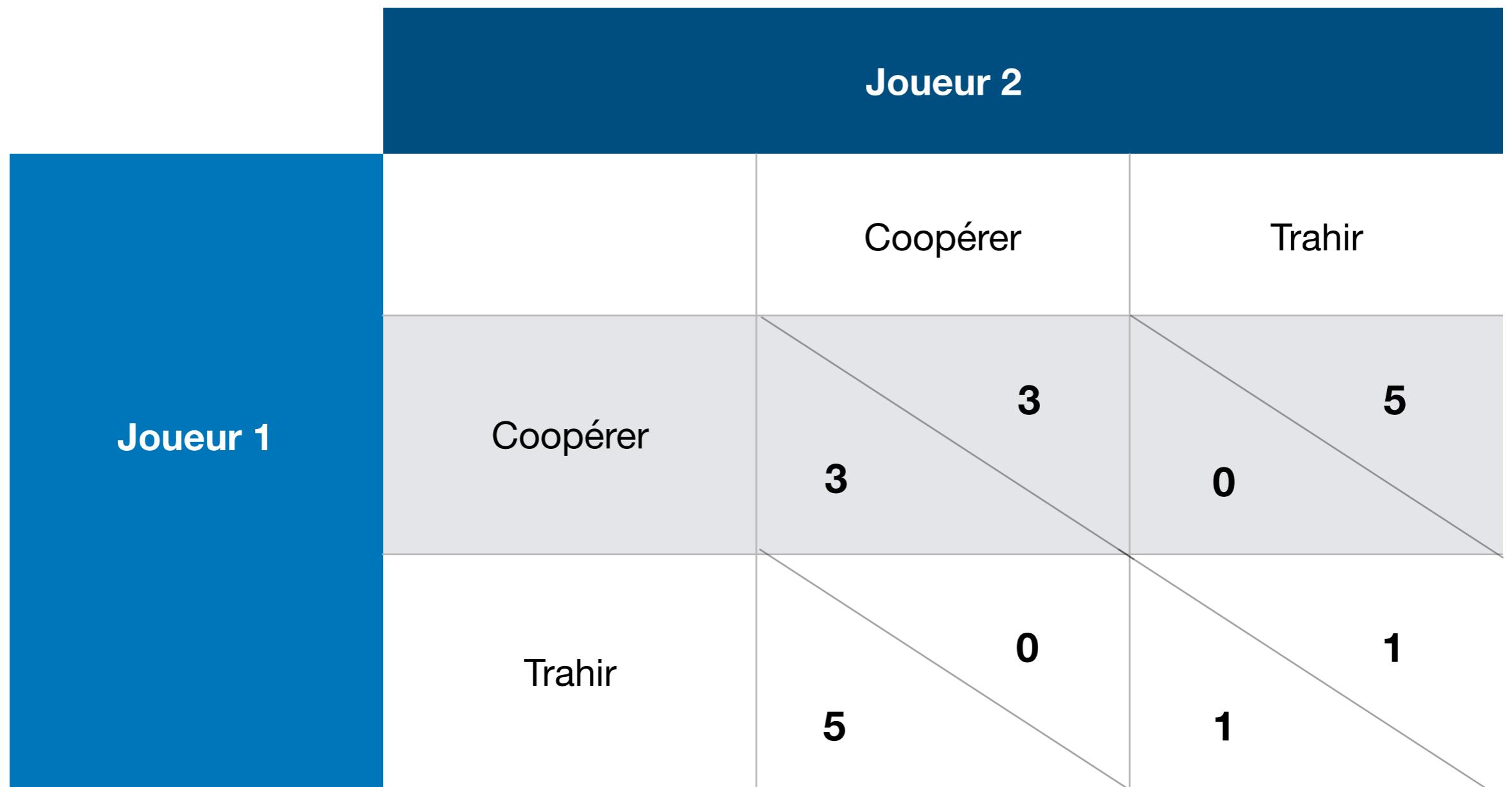
**Recherche de caractéristiques d'une stratégie robuste par introduction d'une classe de stratégies**

- Le dilemme itéré du prisonnier
- Les stratégies probabilistes
- Les stratégies players

# Situation de dilemme

		Bonnie	
		Se taire	Dénoncer son acolyte
Clyde	Se taire	Prend 6 mois de prison Prend 6 mois prison	Libre de partir Peine maximale, 10 ans de prison
	Dénoncer son acolyte	Peine maximale, 10 ans de prison Libre de partir	5 ans de prison 5 ans de prison

# Modélisation avec des gains



# Modélisation avec des gains

		Joueur 2	
		Coopérer	Trahir
Joueur 1	Coopérer	3	5
	Trahir	0	1

## L'équilibre de Nash

On réfléchit à la place du Joueur 1

Dans tous les cas, le mieux est de trahir

Équilibre de Nash : les deux joueurs trahissent  
C'est la pire issue *collectivement*

## Un jeu à somme non constante

Double coopération : 6 points

Double trahison : 2 points

Coopération/trahison : 5 points

La double coopération est *collectivement* la meilleure solution

# Le dilemme itéré du prisonnier

**Stratégie :**

**Prise de décision dépendant des issues des tours précédents, et permettant de répondre à n'importe quelle situation**

La mémoire des tours précédents



Le nombre d'itérations du dilemme



La stratégie de son adversaire



Gentille : CCCCCCCCCCC...  
Méchante : TTTTTTTTTTTTTTTTT...

## Stratégies moins naïves

Donnant-donnant : coopère au premier tour, puis joue ce que son adversaire a joué au tour précédent.

Majorité soft : coopère au premier tour, puis joue ce que son adversaire a joué en majorité. En cas d'égalité, elle coopère.

Rancunière : coopère jusqu'à ce que son adversaire trahisse, auquel cas elle trahit jusqu'à la fin.

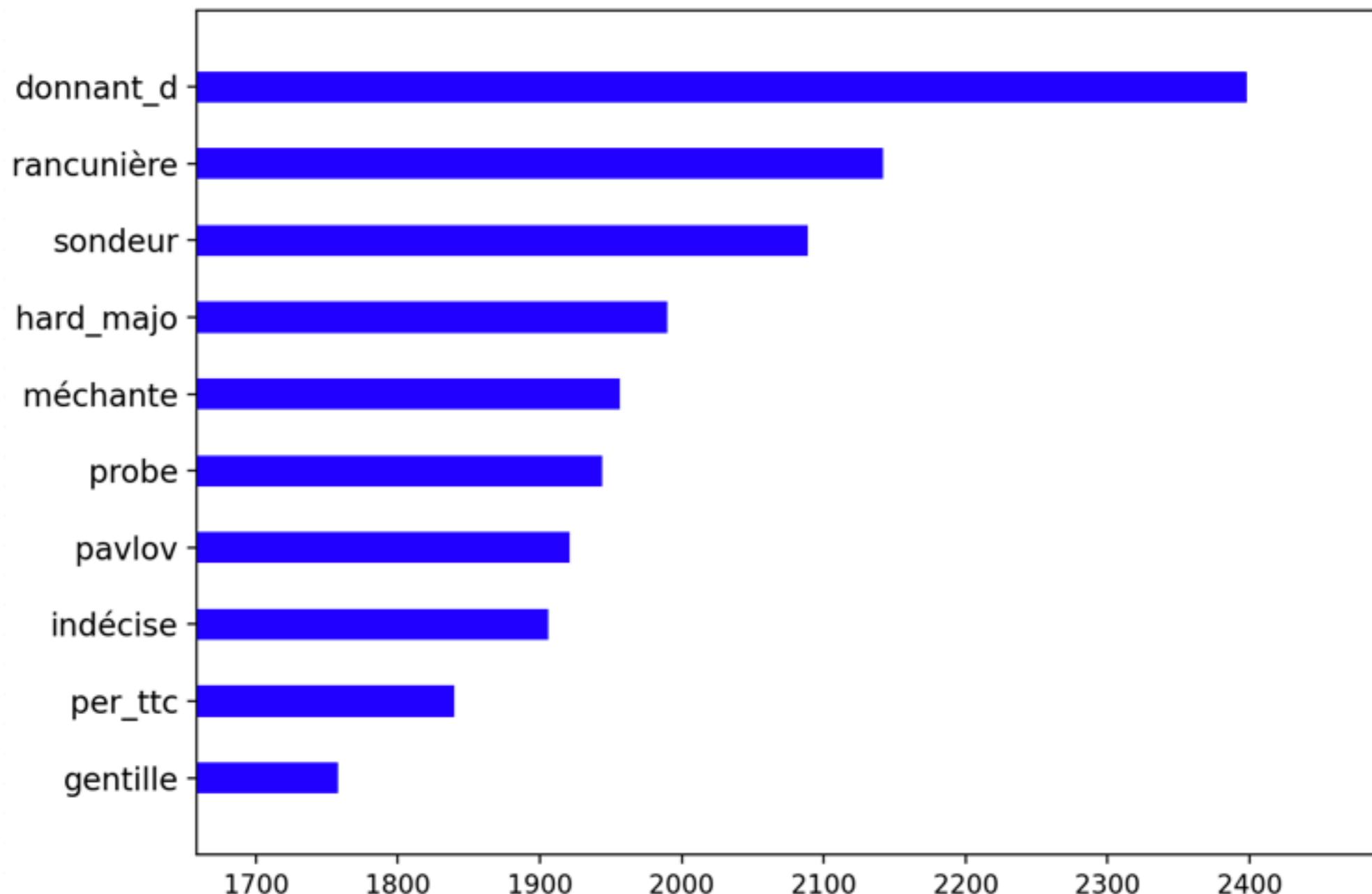
		Joueur 2	
		Coopérer	Trahir
Méchante	Coopérer	3 3	5 0
	Trahir	5 0	1 1

**La meilleure stratégie ?**

**La méchante ne perd aucun match**

# Un groupe de stratégies dans un tournoi

## Résultats du tournoi



Donnant-donnant :  
coopère au premier tour, puis joue ce que  
son adversaire a joué au tour précédent

## Scores de Donnant-donnant\*

Joueur 1	Score	Joueur 2	Score
Donnant-donnant	<b>300</b>	Donnant-donnant	<b>300</b>
Donnant-donnant	<b>300</b>	Rancunière	<b>300</b>
Donnant-donnant	<b>299</b>	Sondeur	<b>299</b>
Donnant-donnant	<b>250</b>	Majorité dure	<b>250</b>
Donnant-donnant	<b>99</b>	Méchante	<b>104</b>
Donnant-donnant	<b>104</b>	Probe	<b>109</b>
Donnant-donnant	<b>300</b>	Pavlov	<b>300</b>
Donnant-donnant	<b>248</b>	Indécise	<b>253</b>
Donnant-donnant	<b>198</b>	Périodique_TTC	<b>203</b>
Donnant-donnant	<b>300</b>	Gentille	<b>300</b>

\* pour 100 itérations du dilemme

## classement\*

Stratégie	Points
Donnant-donnant	<b>2398</b>
Rancunière	<b>2142</b>
Sondeur	<b>2089</b>
Majorité dure	<b>1990</b>
Méchante	<b>1956</b>
Probe	<b>1944</b>
Pavlov	<b>1921</b>
Indécise	<b>1906</b>
Périodique_TTC	<b>1840</b>
Gentille	<b>1758</b>

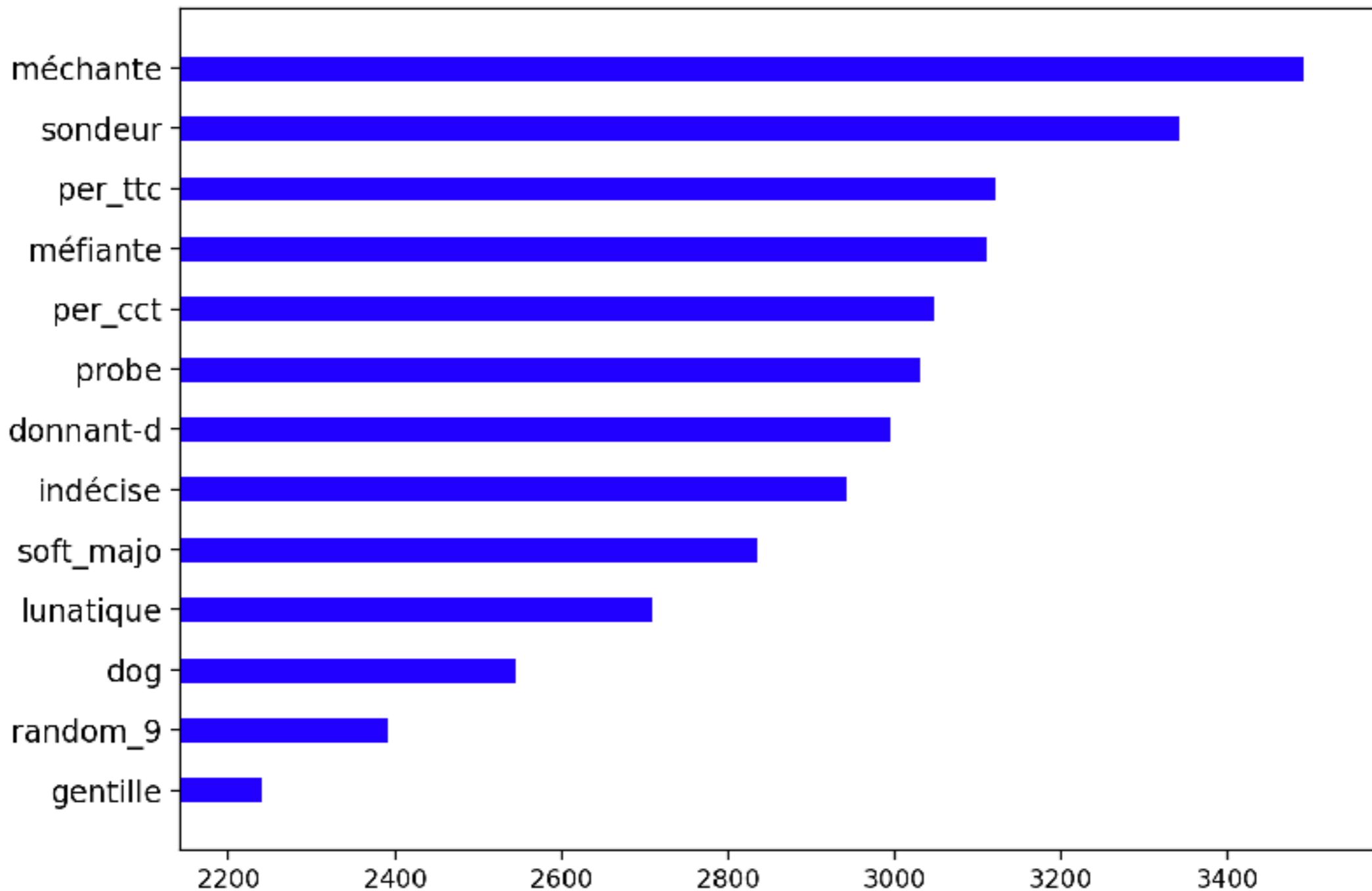
\* pour 100 itérations du dilemme

Un jeu à somme NON constante :

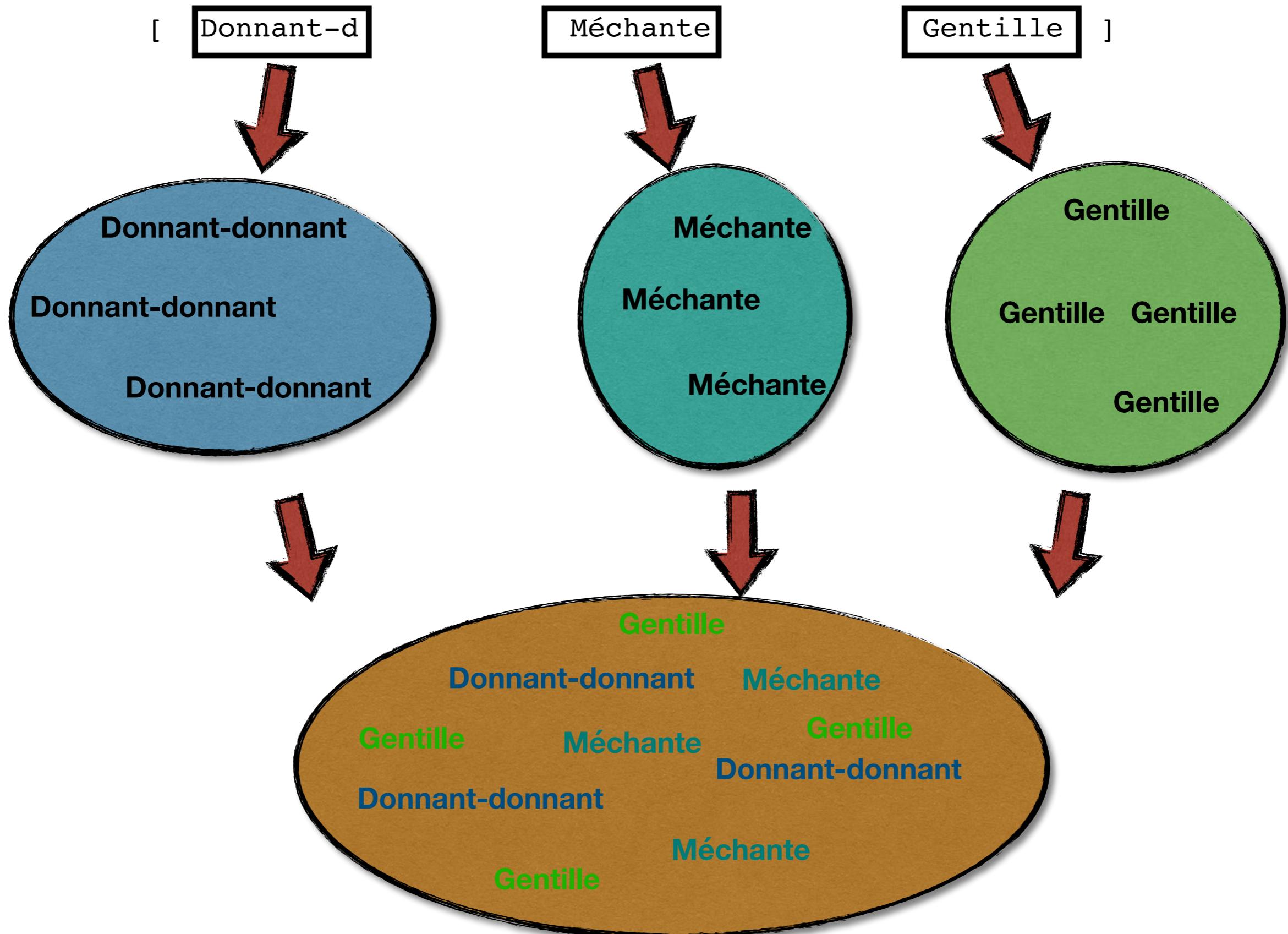
Gagner tous ses matchs  
 <≠>  
 Marquer beaucoup de points

# Un autre groupe de stratégies

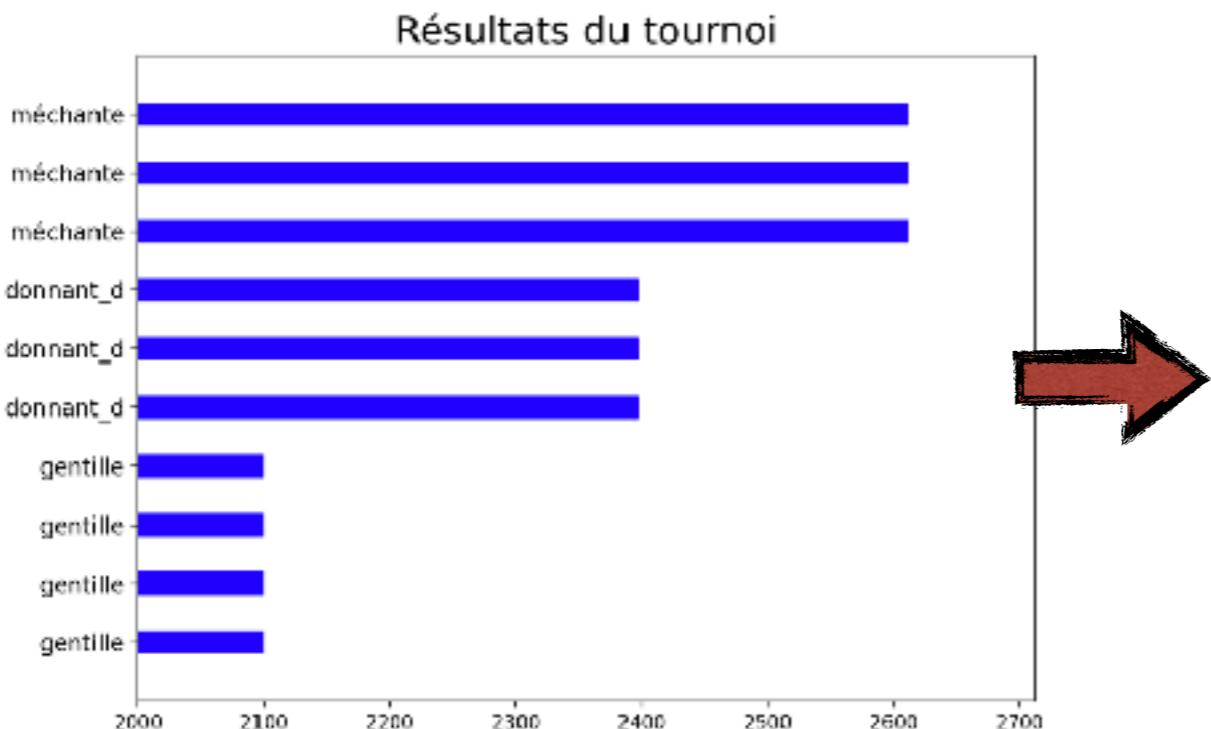
## Résultats du tournoi



# Le jeu de l'évolution

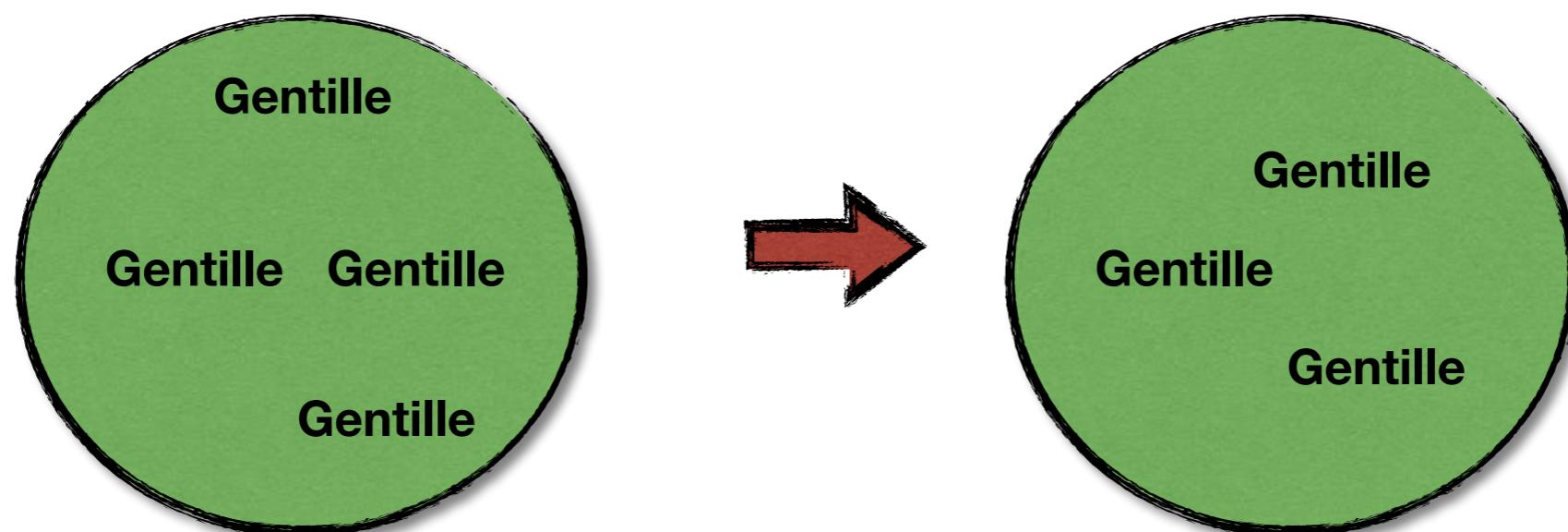


# Le jeu de l'évolution

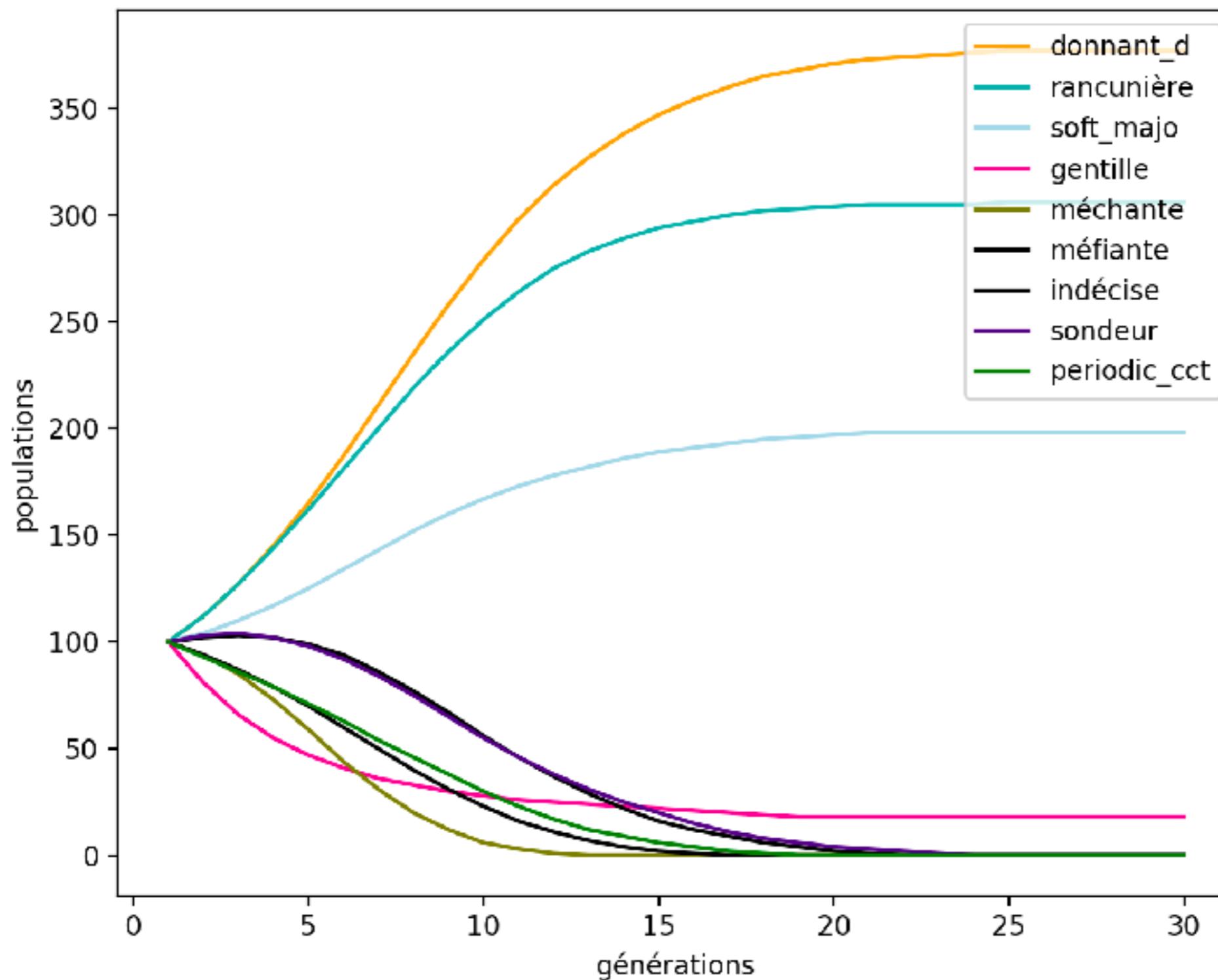


*Génération suivante:*

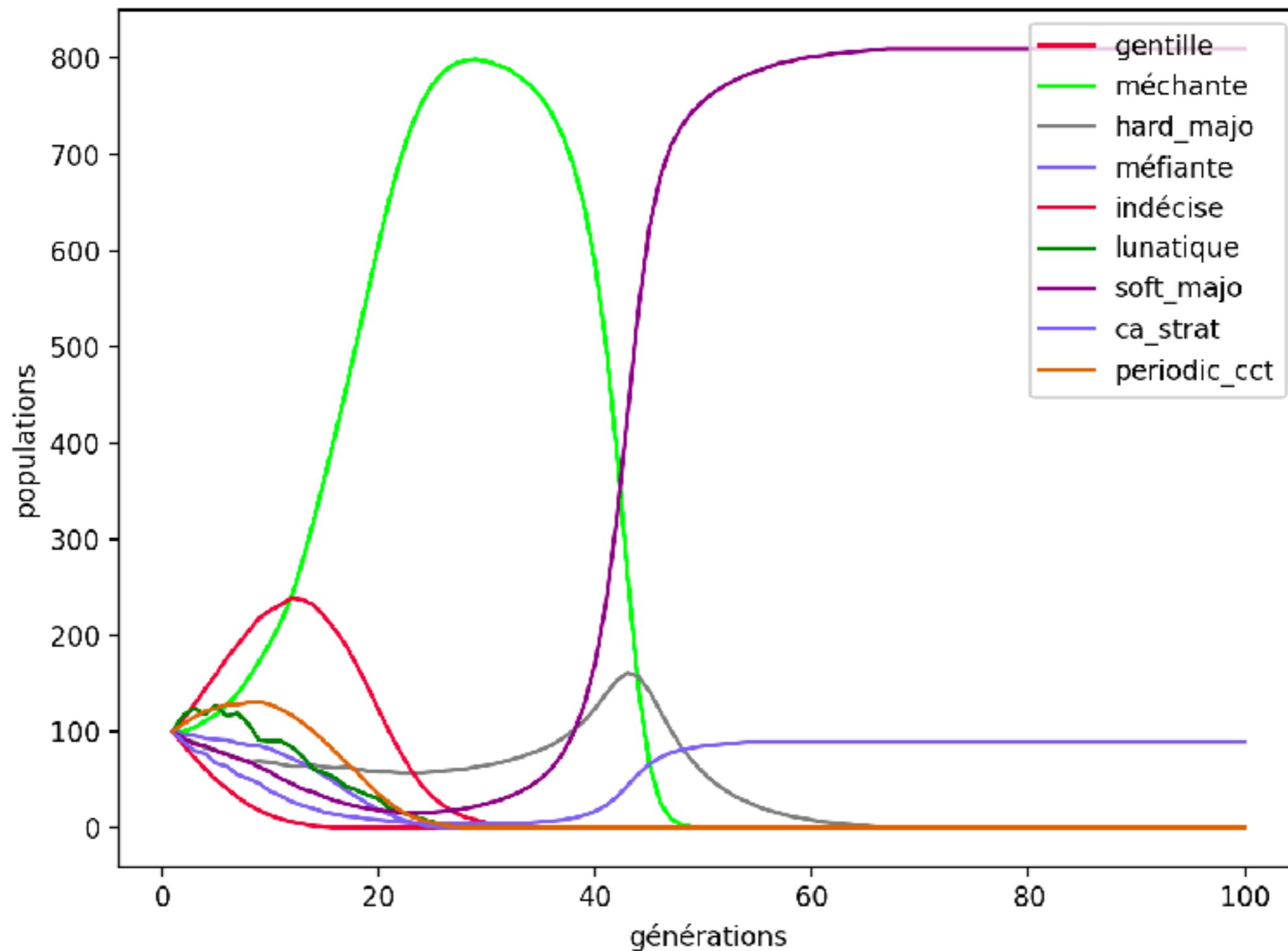
$$pop_{equipe} = \frac{G_{equipe}}{G_{total}} \times pop_{totale}$$



# Résultat jeu de l'évolution



# Autre phénomène du jeu de l'évolution



## **3 critères pour une stratégie robuste établis par Robert Axelrod**

Punir son adversaire quand celui-ci trahit

Ne pas prendre l'initiative de trahir

Coopérer avec soi-même

# Les stratégies probabilistes

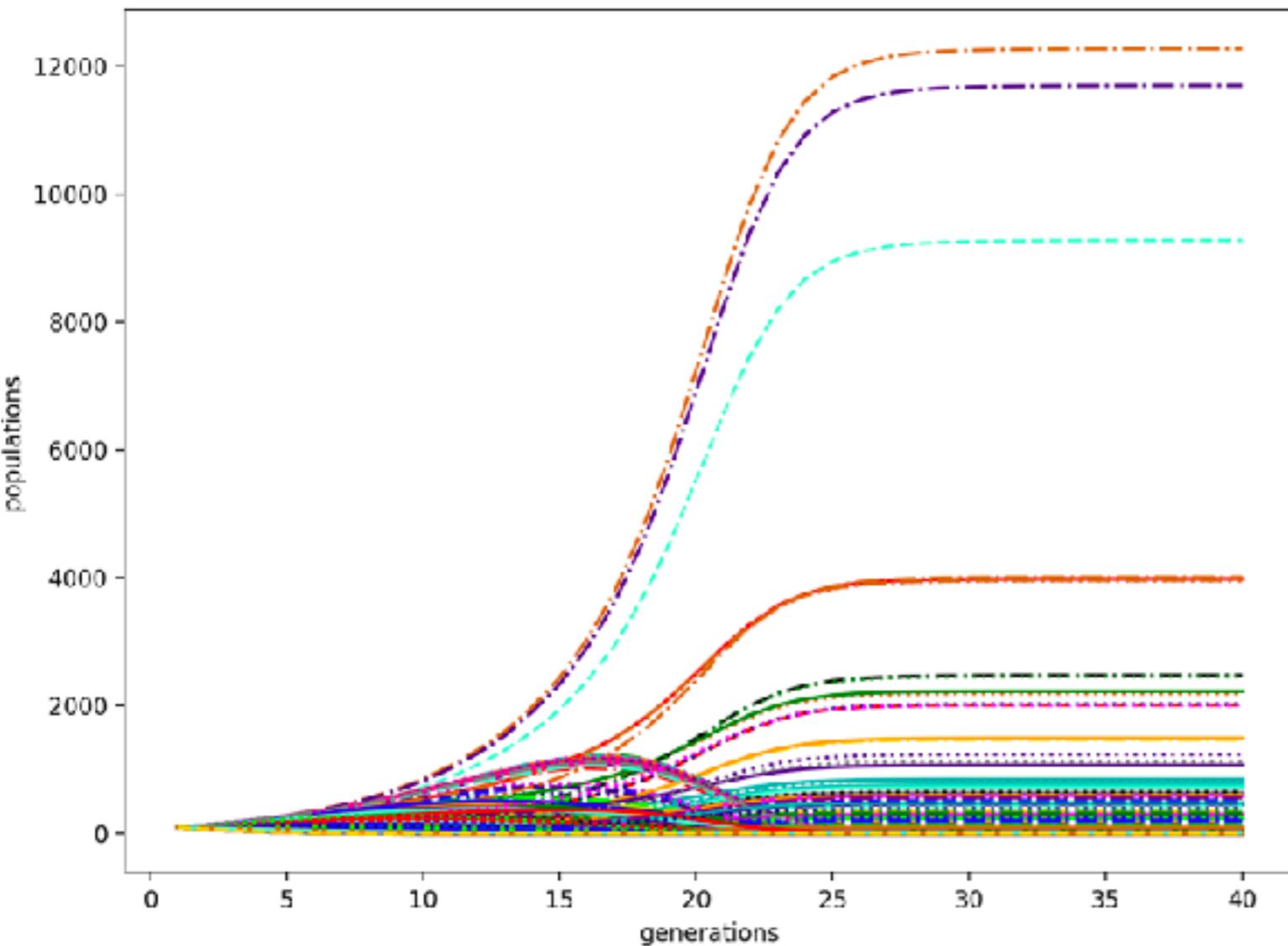
Au tour n-1	proba(q1, q2, q3, q4)		
	Coopérer	Trahir	
proba(p1, p2, p3, p4)	Coopérer	<b>A</b>	<b>B</b>
	Trahir	<b>C</b>	<b>D</b>

Issue	Probabilité de J1
A	p1
B	p2
C	p3
D	p4

Au tour n	proba(q1, q2, q3, q4)		
	Coopérer	Trahir	
proba(p1, p2, p3, p4)	Coopérer	pi <i>qi</i>	pi(1-q <i>i</i> )
	Trahir	q <i>i</i> (1-p <i>i</i> )	(1-p <i>i</i> )(1-q <i>i</i> )

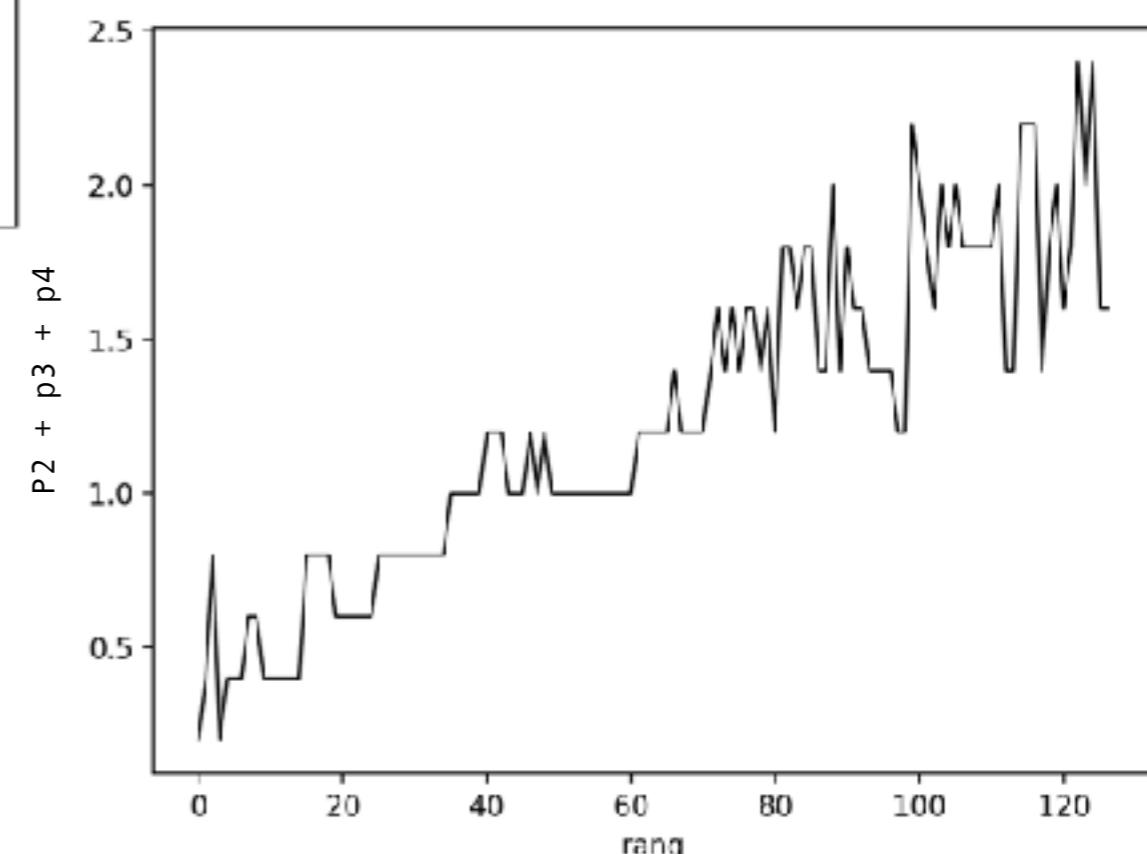
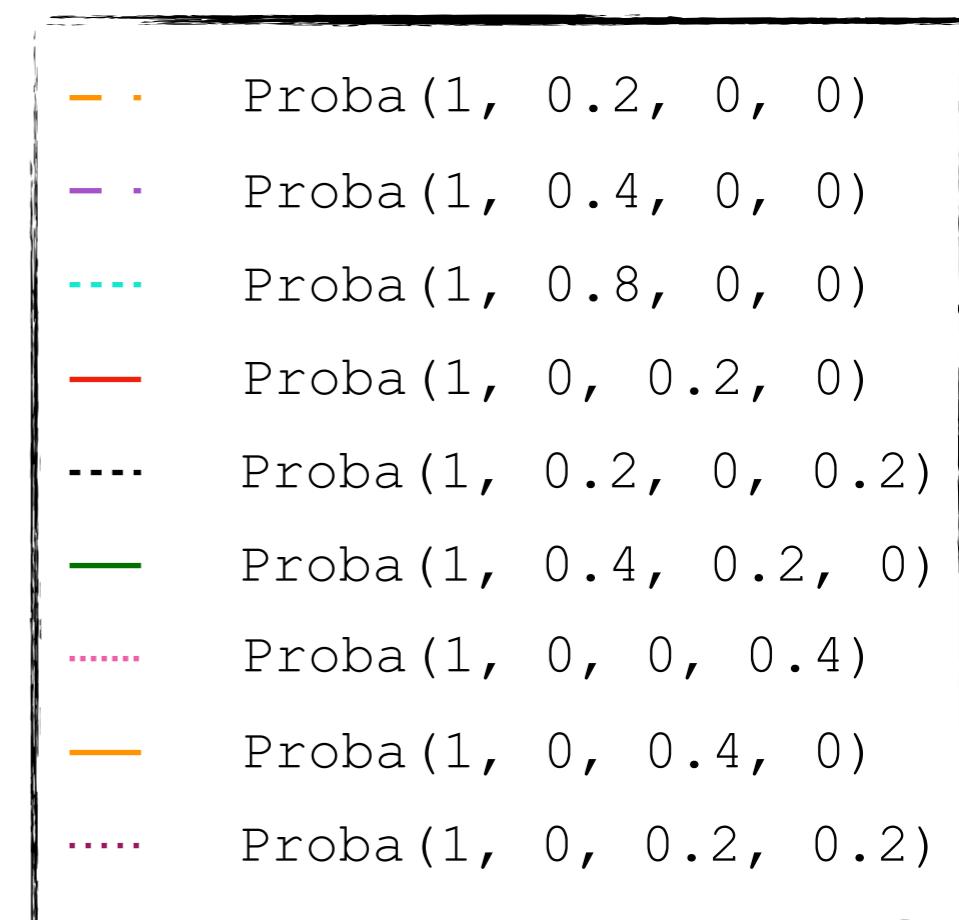
# Étude de Delahaye

$[0, 1] \rightarrow \{0, 1/5, 2/5, 3/5, 4/5, 1\}$



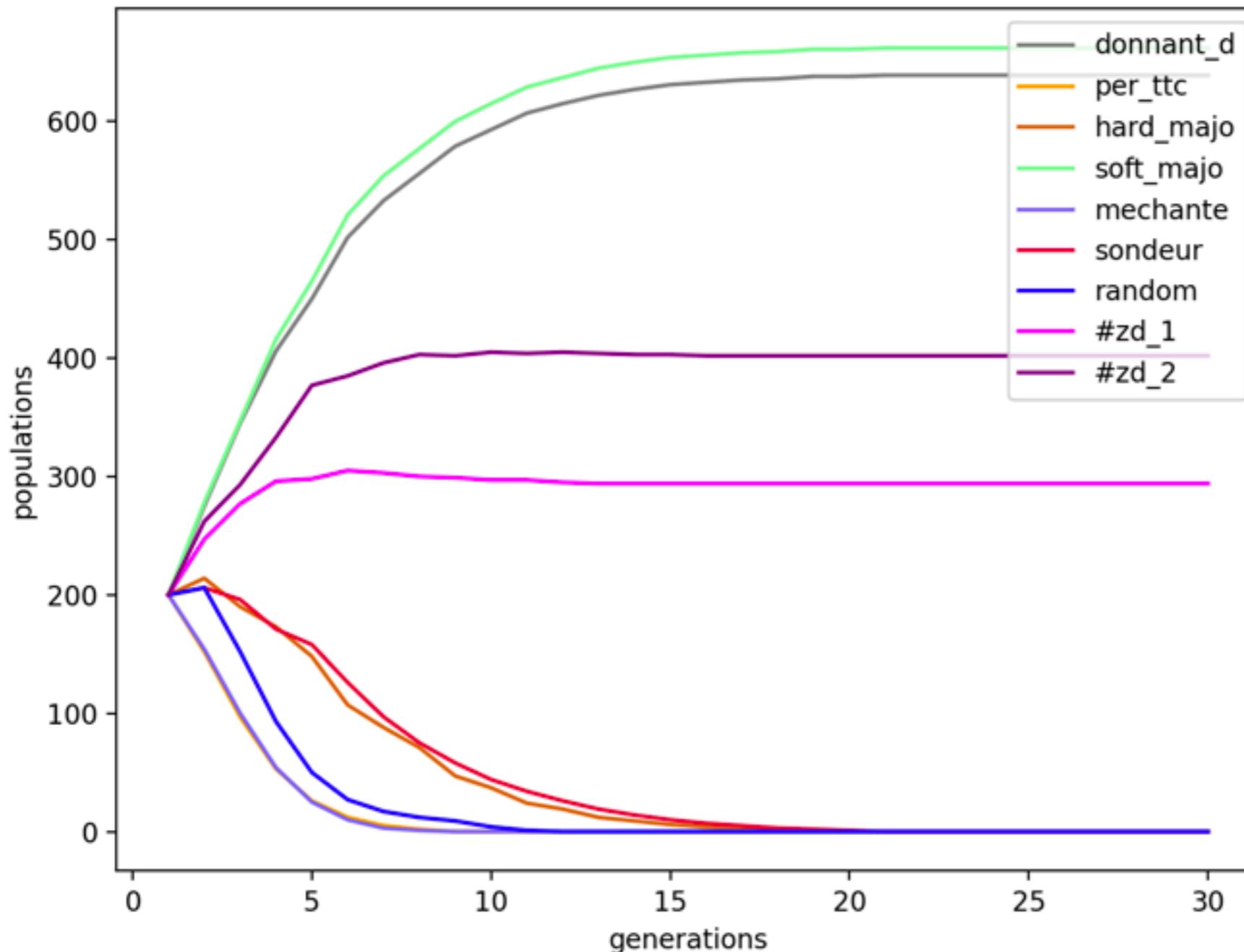
**1000 participantes  
127 survivantes  
100% dont  $p_1 = 1$   
5% des 'mortes' ont  $p_1 = 1$**

Les 9 premières



# Résultats sur les stratégies probabilistes

Critères de robustesse d'une stratégie probabiliste établis par Jean-Paul Delahaye  
 $p_1 = 1$   
 $0 < p_2 + p_3 + p_4 \leq 1$



# Les stratégies ‘players’

**player(x, y, z)**

x : taux de ‘gentillesse’

y : rancune

z : réactivité au dernier coup de l’adversaire

$$coup = x - y \cdot \frac{N_{Trahisons\ adverses}}{N_{Iteration}} + \delta_{C,T} \cdot z$$

Si coup > 0 :  
Player coopère  
Si coup ≤ 0 :  
Player trahit

$\delta_{C,T} = 1$

Si j2 a coopéré

$\delta_{C,T} = -1$

Si j2 a trahi

## Propriétés de la classe players

$$coup = x - y \cdot \frac{N_{Trahisons\ adverses}}{N_{Iteration}} + \delta_{C,T} \cdot z$$

Donnant-donnant :  $\text{player}(0.1, 0, 1)$

Gentille :  $\text{player}(1, 0, 0)$

Méchante :  $\text{player}(-1, 0, 0)$

Rancunière :  $\text{player}(1, \infty, 0)$

Pour tout  $\text{player}(x, y, z)$ ,  
Pour tout  $k$  réel positif,

$\text{player}(kx, ky, kz) = \text{player}(x, y, z)$

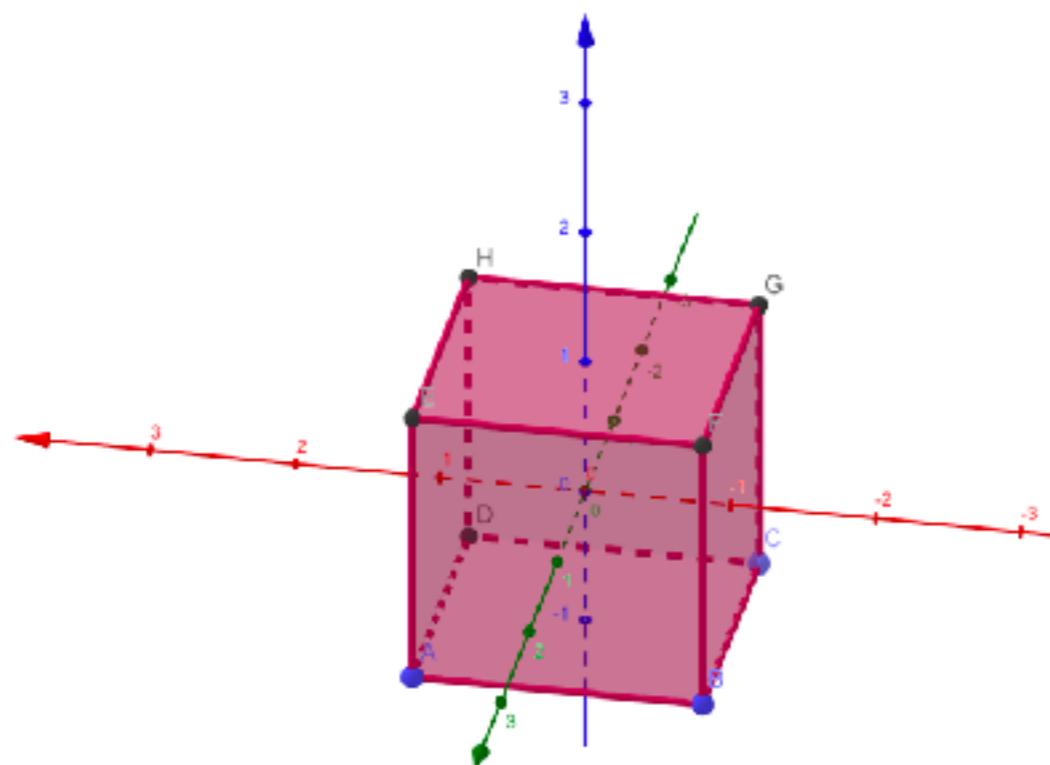
Tout les  $\text{player}(x, y, z)$  tel que

$$z > |x| + |y|$$

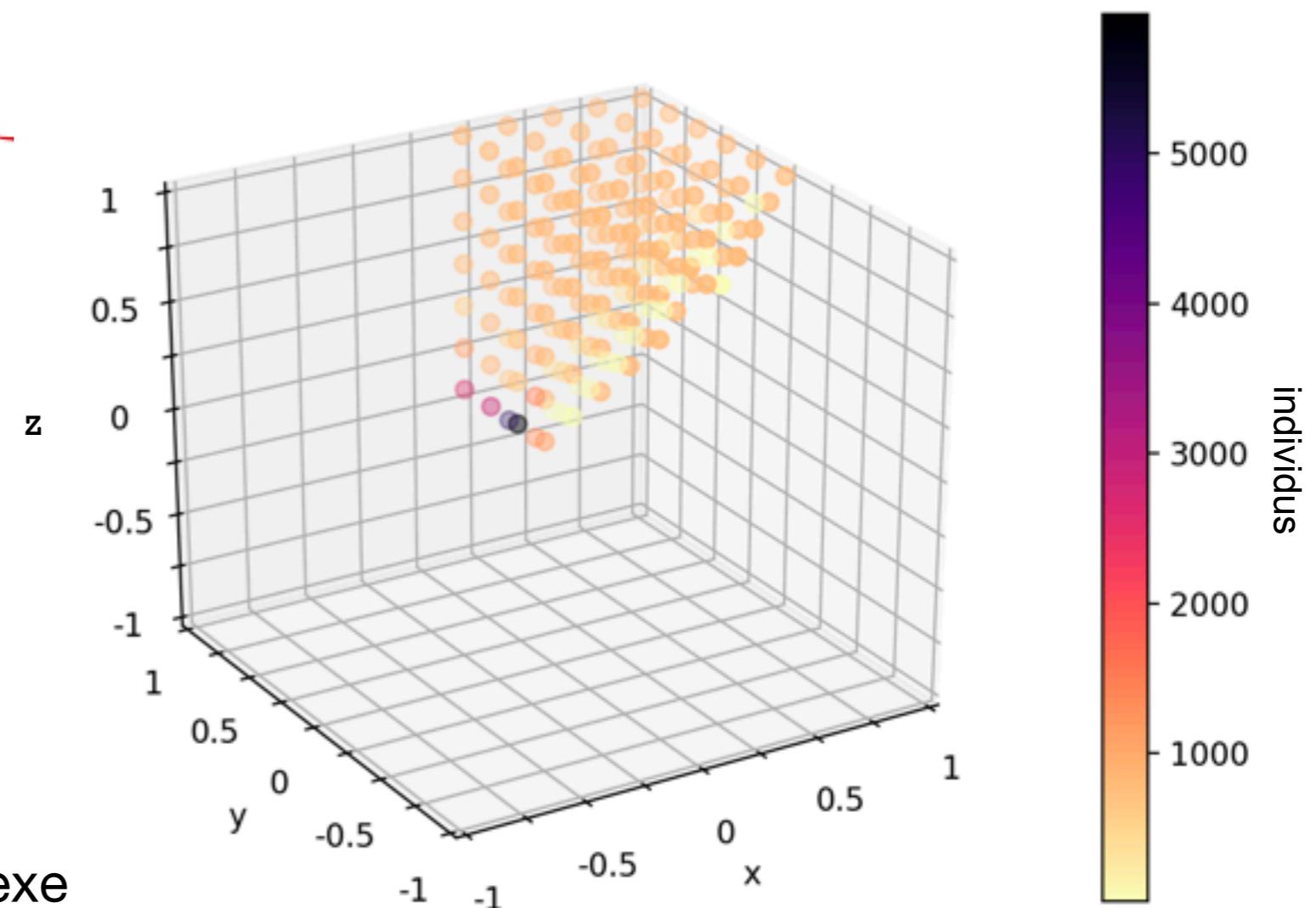
est un donnant-donnant

# La méthode de la discréétisation

$x, y, z$  choisis dans  $[-1, 1]$

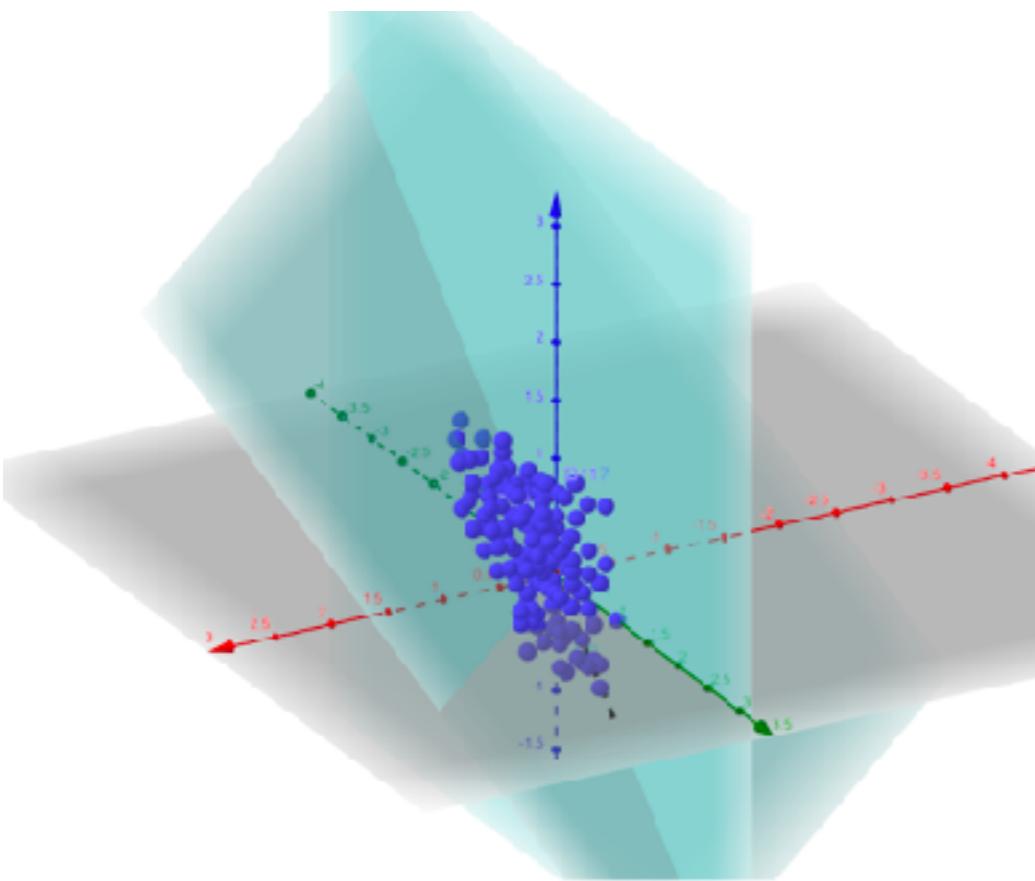
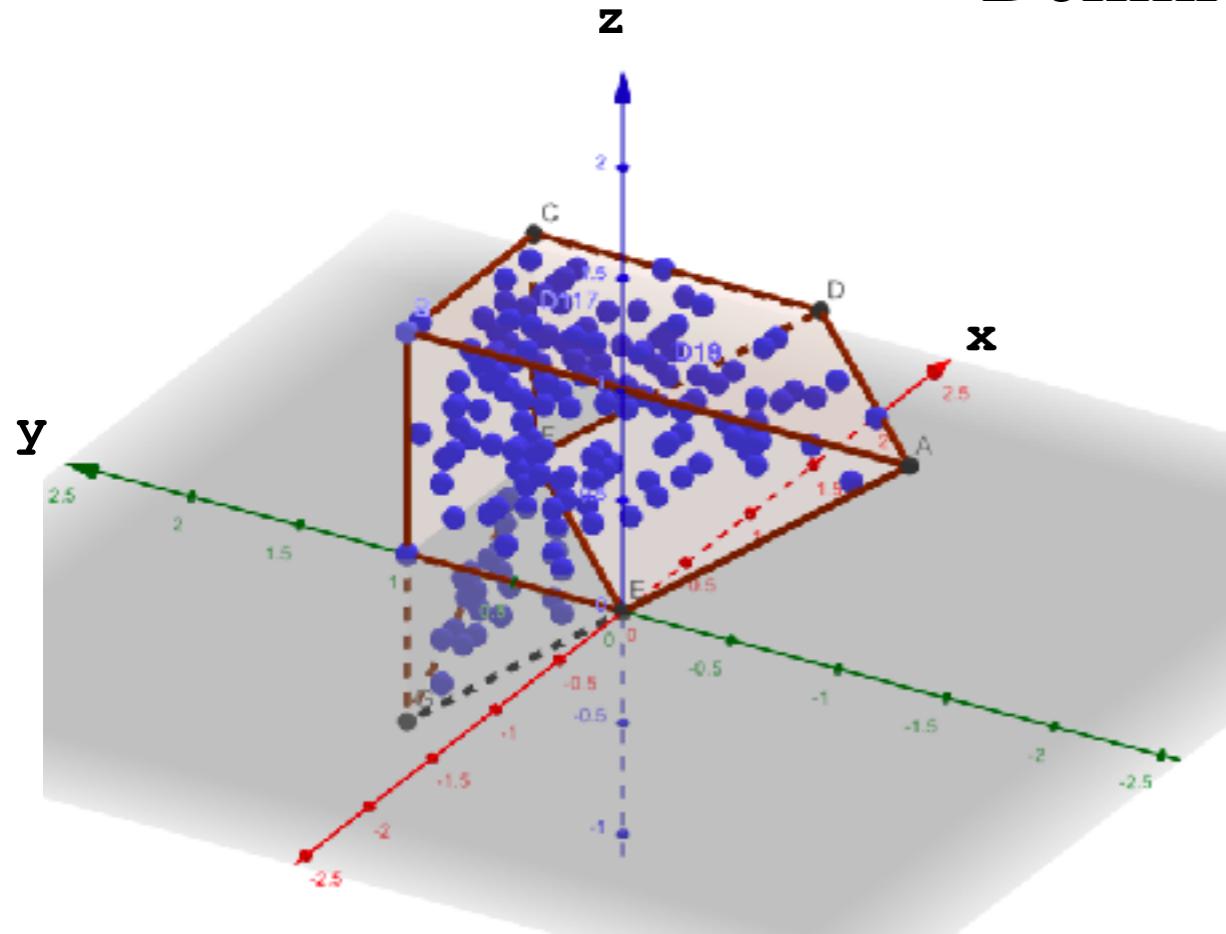


Une première discréétisation grossière

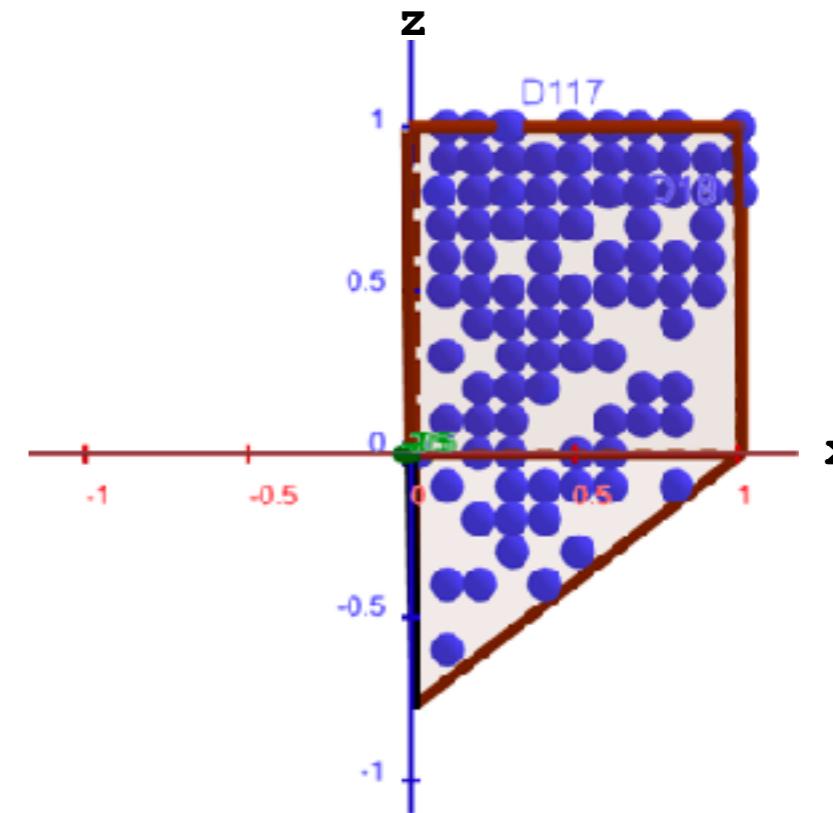


Les stratégies survivantes se regroupent dans un domaine convexe

# Délimitation du domaine



Projection sur  $y = 0$



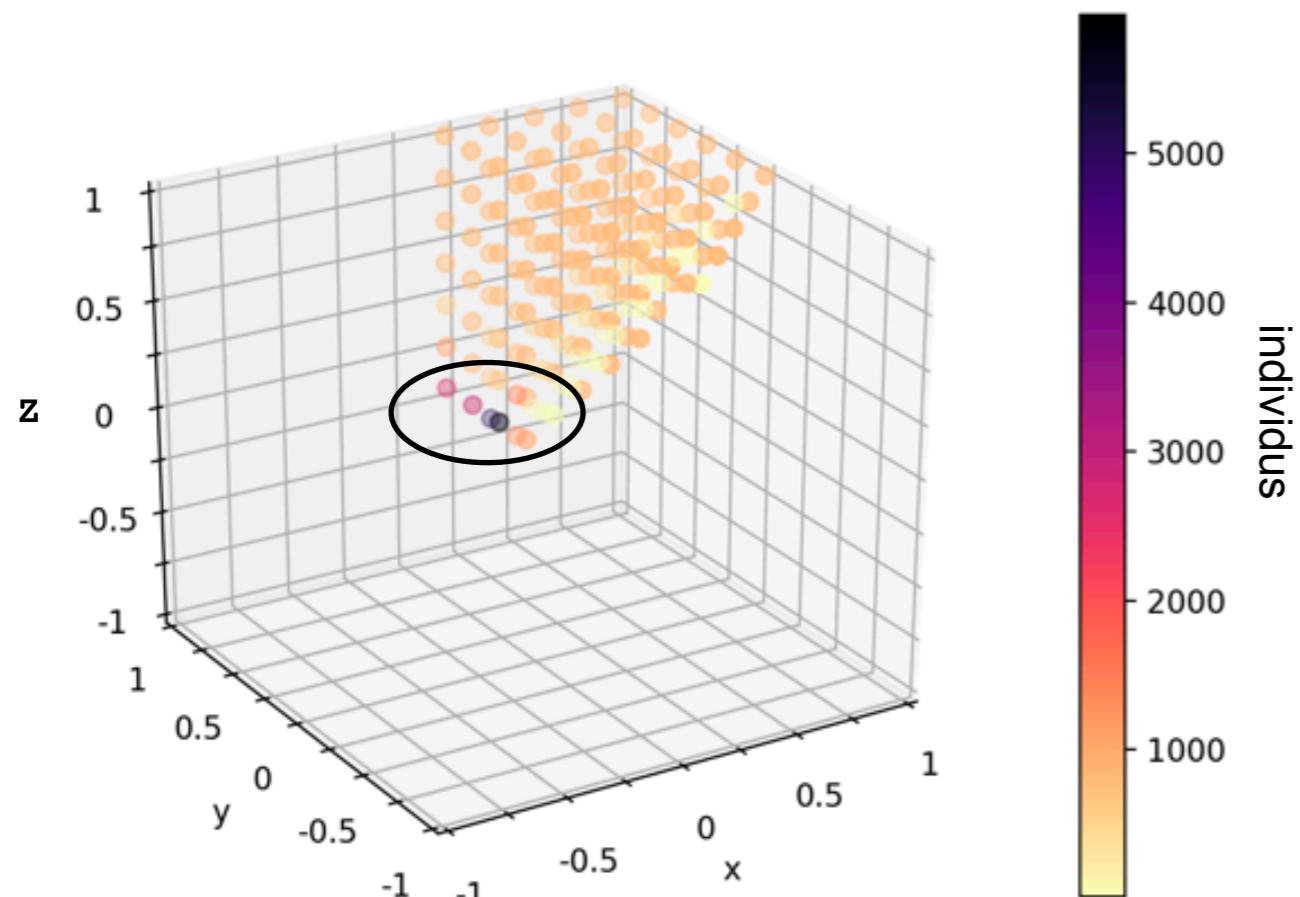
$$x - y - z \leq 0$$

$$x \geq 0$$

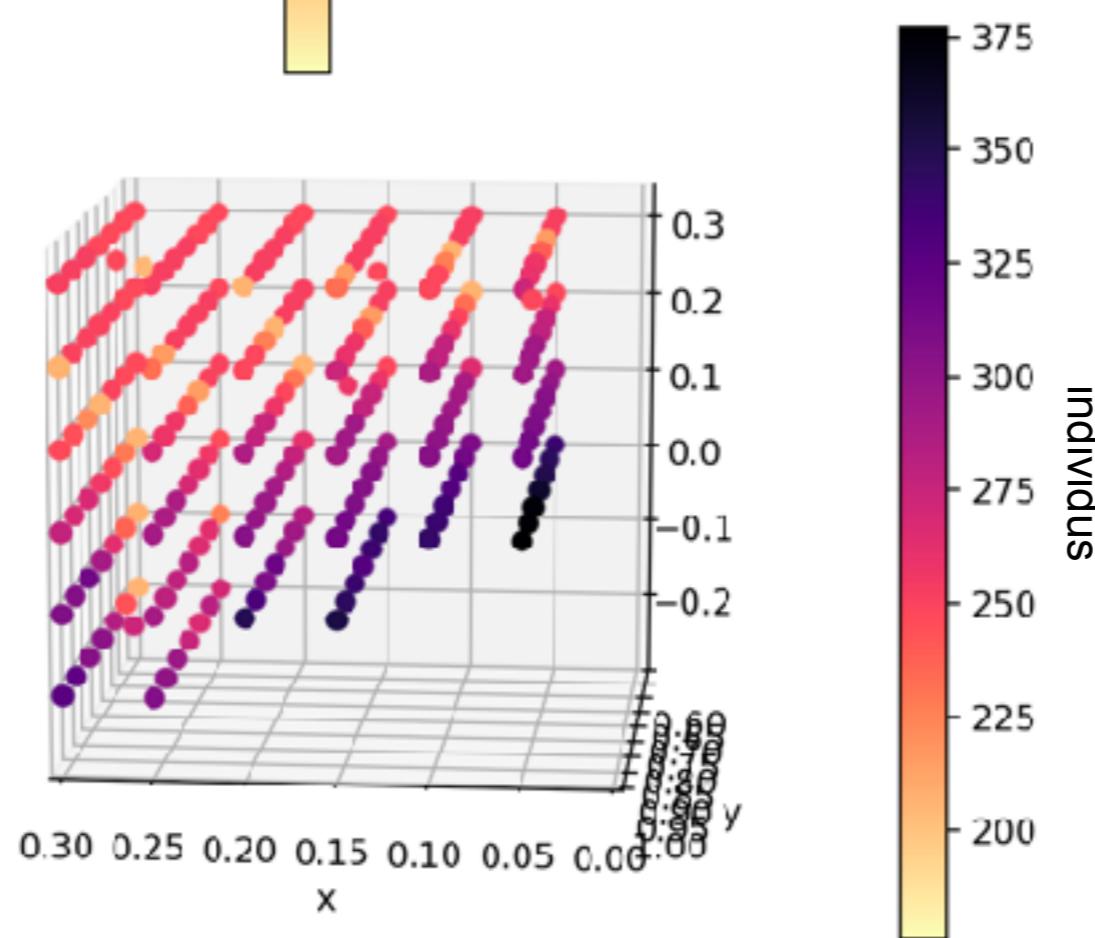
$x \geq 0$  force une coopération au premier tour

**Le domaine est trop grand pour tirer une conclusion sur les caractéristiques de ces stratégies**

# Un zoom sur le maximum d'individus

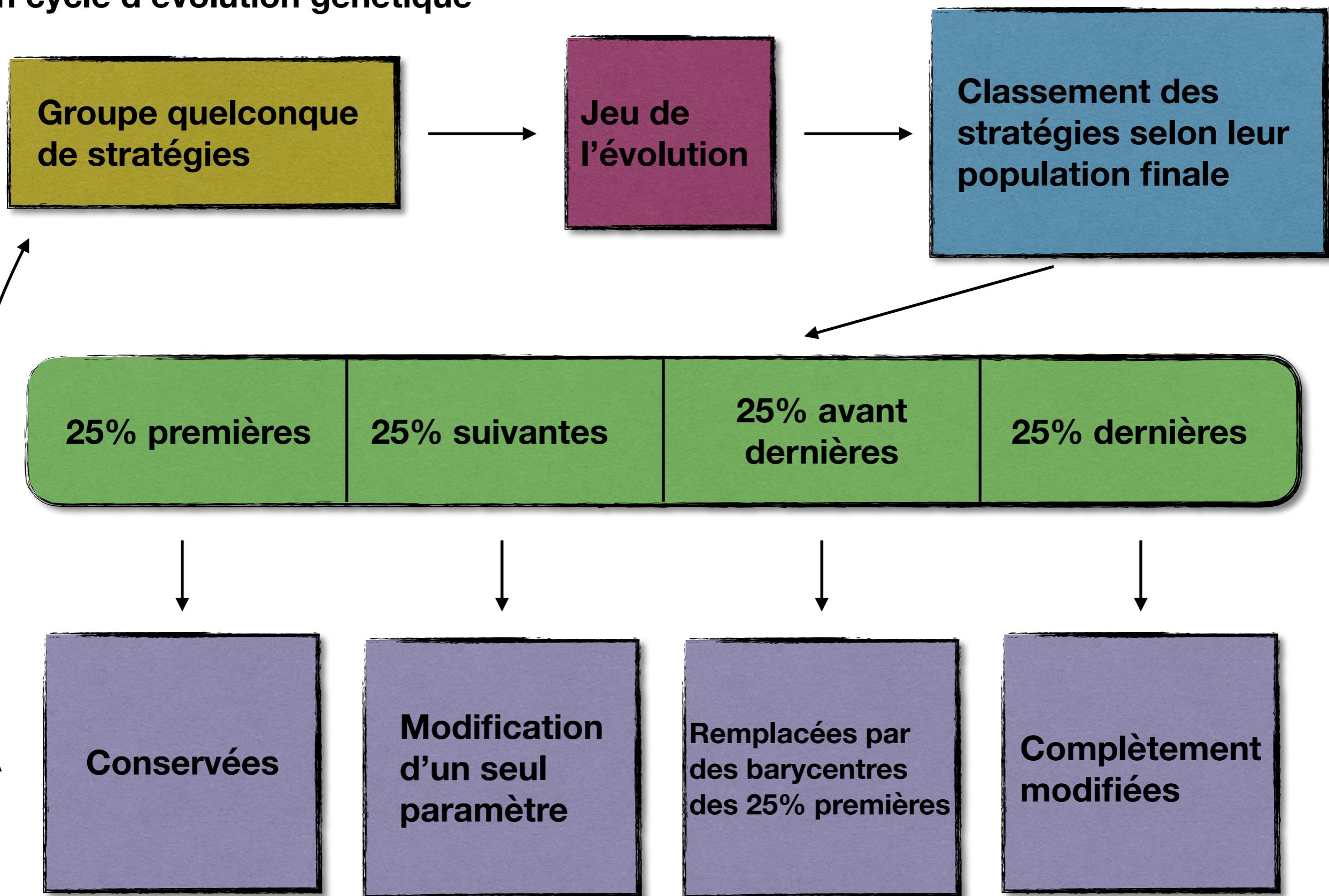


**On effectue un zoom  
On discrétise un parallélépipède  
On rajoute des stratégies  
aléatoirement dans le cube**

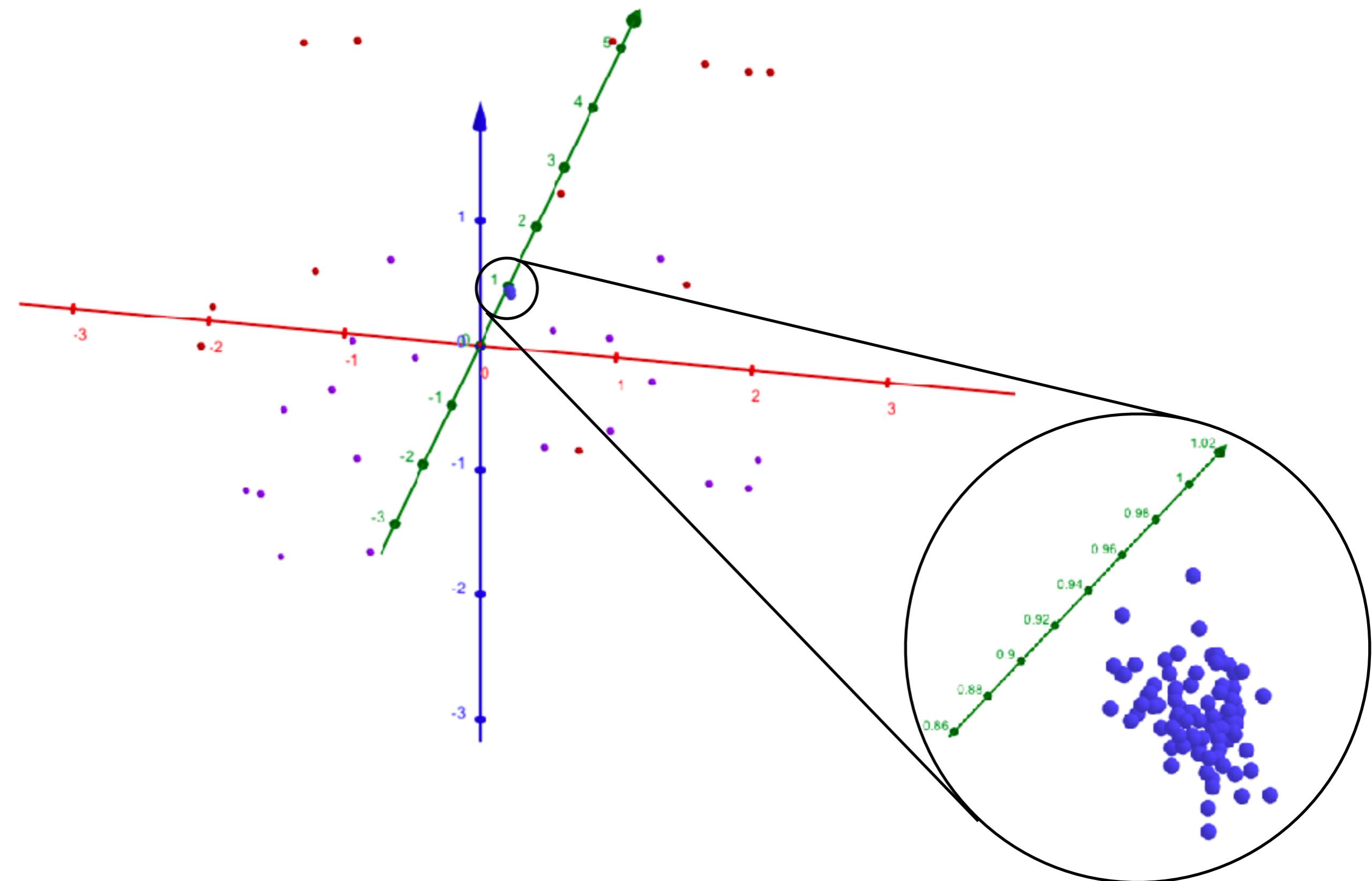


# La méthode de l'évolution génétique

## Un cycle d'évolution génétique



# Point de convergence de l'algorithme génétique



# Interprétation de la stratégie

```
B = player(0.0284, 0.9205, -0.0147)
```

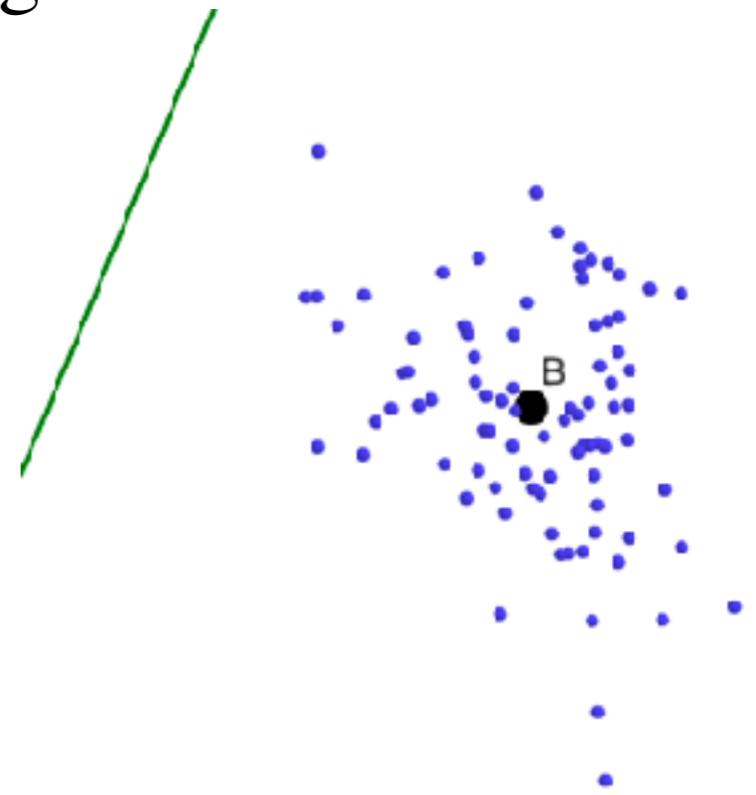
$$coup = x - y \cdot \frac{N_{Trahisons\ adverses}}{N_{Iteration}} + \delta_{C,T} \cdot z$$

x et z sont très proches de 0

Z est négatif

Y est proche de 1

$$\frac{y}{x} = 32.41$$

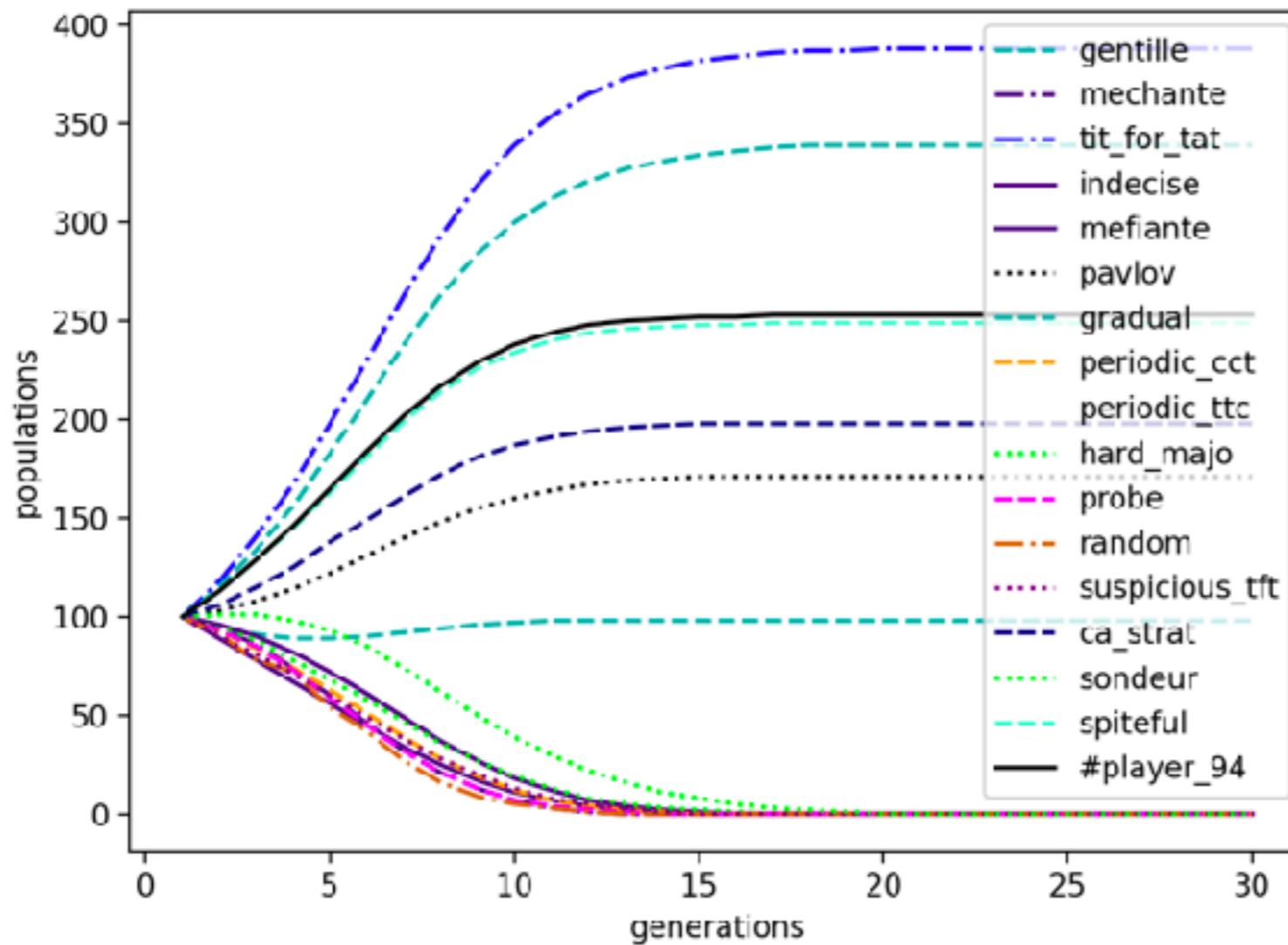


## Interprétation de cette stratégie

Pour N\_iteration faible,  
les punitions sont lourdes  
pour N\_iteration grand,  
les punitions sont moins importantes

Une confiance s'installe : plus elle connaît son adversaire et plus elle lui pardonne facilement

# Conclusion, ouverture



B dépend de N\_iteration

Étudier la variation de B  
selon N\_iteration

```

import random as rd
import numpy as np
import matplotlib.pyplot as plt

""" CLASSES DES INDIVIDUS """

class strategies :

    def __init__(self, strat, stratname, Nindiv, lens) :
        self.strategy = strat
        self.stratname = stratname

        self.gain = 0
        self.gain_list = [0] * lens # gain_liste[k] est son gain total contre l'individu k
        self.gain_final = 0

        self.memory = [[], []] # il se verra toujours comme un J1
        self.param = [] # paramètre qui est défini et utilisé dans les fonctions
        self.individuals = Nindiv #nombre d'individus dans le groupe

    def play(self) :
        return self.strategy(self.memory, self.param) #appelle une des fonctions

    def reset(self) :
        self.memory = [[], []] #après chaque match, dans match(), remet à 0 le memory
        self.param = []

    def repop(self, gaintot) : #pour redéfinir la population (self.Nindiv) du jeu
        self.individuals = int(self.gain_final * Ntot / gaintot)

class ZD :

    last_game = [[True, True], [True, False], [False, True], [False, False]] # liste de 4 stratégies possibles
    players_created = 0 #pour nommer les stratégies sur leur numéro de création

    def __init__(self, parameters, stratname, Nindiv, lens) :
        ZD.players_created +=1

        self.strategy = parameters
        self.stratname = '#zd_'+str(ZD.players_created)
        self.gain = 0

```

```

        self.gain_list = [0]*lens
        self.gain_final = 0

        self.memory = [[], []]
        self.individuals = Nindiv

    def recognize(self, coopere) : #coopere est de la forme [dernier coup de zD, ...]
        for k in range(4) : # 4 = len(ZD.last_game)
            if ZD.last_game[k] == coopere :
                return self.strategy[k] #renvoie la probabilité de coopérer en fonction de l'histoire

    def play(self) :
        if len(self.memory[0]) == 0 : #si c'est le premier tour, joue la coopération
            return True
        last_play = [self.memory[0][0], self.memory[1][0]] #on sauvegarde le dernier tour
        self.memory = [[], []] #et on supprime le reste
        p = rd.random() #on tire au sort un coup qui va être joué
        pk = self.recognize(last_play) #on reconnaît l'issue du dernier tour et on calcule la probabilité
        if p > pk : #si p > pk, alors il trahit, sinon il coopère
            return False
        else :
            return True

    def reset(self) :
        self.memory = [[], []]

    def repop(self, gaintot) :
        self.individuals = int(self.gain_final * Ntot / gaintot)

class player :

    players_created = 0

    def __init__(self, vector, stratname, Nindiv, lens) : #vector est un array de 4 stratégies
        player.players_created +=1

        self.strategy = vector
        self.stratname = '#player_'+str(player.players_created)

        self.gain = 0
        self.gain_list = [0] * lens # gain_liste[k] est son gain total contre l'individu k
        self.gain_final = 0

        self.memory = [[], []] #il se verra toujours comme un J1
        self.individuals = Nindiv #nombre d'individus dans le groupe

        self.marker = 0 #dans l'algorithme génétique, sert à voir combien de bactéries sont créées

```

```

def play(self) :
    l = len(self.memory[0])
    if l == 0 :
        b = self.strategy[0]
    else :
        if self.memory[1][-1] :
            r = 1
        else :
            r = -1
        b = self.strategy[0] - (l-counter_true(self.memory[1])) * self.strat
    if b > 0 :
        return True
    else :
        return False

def reset(self) :
    self.memory = [[], []] #après chaque match, dans match(), remet à 0 la

def repop(self, gaintot) : #pour redéfinir la population (self.Nindiv) du j
    self.individuals = int(self.gain_final * Ntot / gaintot)

"""
DEBUT STRATEGIES """
def gentille(coups, param) : #all_c
    return True

def mechante(coups, param) : #all_t
    return False

def tit_for_tat(coups, param) :
    if len(coups[0]) == 0 :
        return True
    else :
        return coups[1][-1]

def mefiaente(coups, param) : # joue périodiquement tc en commençant par t
    if len(coups[0]) % 2 == 0 :
        return False
    else :
        -

```

```

        return True

def indecise(coups, param) : # joue périodiquement ct en commençant par c
    if len(coups[0]) % 2 == 0 :
        return True
    else :
        return False

def random(coups, param) : #joue aléatoirement
    a = rd.randint(1,2)
    if a == 1 :
        return True
    else :
        return False

def soft_majo(coups, param) : #joue ce que l'autre a joué en majorité, et joue
    def scount(coups) :
        c = 0
        for k in coups[1] :
            if k :
                c += 1
        if c >= (len(coups[1]))/2 :
            return True
        else :
            return False

    if len(coups[0]) == 0 or scount(coups) :
        return True
    elif not scount(coups) :
        return False

def spiteful(coups, param) : #joue c jusqu'à ce que l'autre trahisse, puis trahit
    if counter_true(coups[1]) != len(coups[0]) :
        return False
    else :
        return True

def sondeur(coups, param) : # aux 3 premiers coups il joue tcc, puis t tout le
    l = len(coups[0])
    if l == 0 :
        return False
    if l == 1 or l == 2 :
        return True
    if coups[1][1] and coups[1][2] :
        return False
    -

```

```

else :
    return tit_for_tat(coups, param)

def periodic_cct(coups, param) : #joue cooperer cooperer trahir périodiquement
l = len(coups[0])%3
if l == 0 or l == 1:
    return True
else :
    return False

def periodic_ttc(coups, param) : #joue trahir trahir coopérer coopérer périodiquement
l = len(coups[0])%3
if l == 0 or l == 1:
    return False
else :
    return True

def hard_majo(coups, param) : #trahit au premier coup ou si l'adversaire a majorité
def hcount(coups) :
    c = 0
    for k in coups[1] :
        if k :
            c += 1
    if c > (len(coups[1]))/2 :
        return True
    else :
        return False

    if len(coups[0]) == 0 or not hcount(coups) :
        return False
    elif hcount(coups) :
        return True

def probe(coups, param) : #coopère au premier coup puis coopère seulement si l'autre a trahi
if len(coups[0]) == 0 :
    return True
elif coups[-1] == [True,True] or coups[-1] == [False,False] : #analyse les derniers coups
    return True
else :
    return False

def pavlov(coups, param) :
l = len(coups[0])
if l == 0 :

```

```

        return True
last_played = coups[0][-1]
last_earned = dilemme([last_played, coups[1][-1]])[0]
if last_earned >= 3 :
    return last_played #s'il a gagné plus que 3, il rejoue pareil qu'au tour précédent
else :
    return (last_played+1)%2 #s'il a gagné moins que 3, il joue l'autre coup

def gradual(coups, param) :
"""
param[0] est une liste des coups qu'il a prévu de renvoyer,
il renvoie la donnée qui est en premier dans la liste.
"""
l = len(coups[0])
if l == 0 :
    param.append([[],0])
    return True
elif len(param[0]) != 0 :
    ans = param[0][0]
    del param[0][0]
    return ans
elif len(param[0]) == 0 :
    c = l - counter_true(coups[1])
    if param[1] < c :
        param[1] = c
        param[0] = [False]*(c-1) + [True]*2
        return False
    elif param[1] == c :
        return True

def suspicious_tft(coups, param) :
l = len(coups[0])
if l == 0 :
    return False
else :
    return coups[1][-1]

def ca_strat(coups, param) : #coopère jusqu'à ce que l'autre trahisse n fois
l = len(coups[1])
if l == 0 :
    param.append(5) #threshold
    param.append(False) #reached_threshold
    param.append(0.1) #percent
    return True
else :
    if param[1] :
        return False

```

```

f = l - counter_true(coups[1]) # = counter_false
if f >= param[0] :
    param[1] = True
    return False
if coups[1][-1] :
    param[0] += param[0] * param[2]
return True

def random_9(coups, param) :
    coup = rd.random()
    if coup <= 0.9 :
        return True
    else :
        return False

def dog(coups, param) :
    l = len(coups[0])
    if l <= 2 :
        return False
    if coups[1][0] and not coups[1][1] :
        param.append(False)
    else :
        param.append(True)
    return param[0]

def counter_true(coups) : #compte le nombre de cooperations de l'adversaire
    c = 0
    for k in coups :
        if k :
            c += 1
    return c

def counter_while_false(coups) : #compte le nombre de false de l'adversaire, tant qu'il y a un true
    k = len(coups[1]) #(au premier true, le compteur s'arrête)
    c = 0
    while not coups[0][k-1] :
        if k-1 >= 0 :
            c += 1
            k -= 1
    return c

```

\*\*\*\*\* DILEMME ITERE DU PRISONNIER \*\*\*\*\*

```

def dilemme(coopere): #définira le gain de chacun des deux joueurs. True=coopérant, False=non-coopérant
    #coopere[0] est l'action choisie par J1, coopere[1] est celle de J2, et le gain est une liste [J1,J2]
    if coopere[0] :
        if coopere[1] :
            return [3,3]
        else :
            return [0,5]
    else :
        if not coopere[1] :
            return [1,1]
        else :
            return [5,0]

def partie(player1, player2): #fait s'affronter deux stratégies une seule fois, J1,J2 = player1.play(), player2.play() #on acquiert le coup que va jouer chaque joueur
    gain = dilemme([J1,J2]) #calcule le gain que leur coup implique
    player1.gain += gain[0] #distribue le gain à chacun
    player2.gain += gain[1] #le gain est cumulé, il sera remis à zéro à la fin
    player1.memory[0].append(J1) #la mémoire est aussi mise à jour
    player1.memory[1].append(J2)
    player2.memory[0].append(J2) #l'inversion des listes se fait ici
    player2.memory[1].append(J1)

def match(n,player1,player2) : #n nombre de parties par affrontement.
    for k in range(n) : #n itérations du dilemme par match
        partie(player1,player2)
    player1.reset() #à la fin du match, on reset les attributs des joueurs
    player2.reset()

def tournoi_fast(n, teamL) :
    lenS = len(teamL)
    for k in range(lenS) : #chaque individu de la liste va affronter tous les autres
        for l in range(k, lenS) :
            if k == l : #s'il s'affronte lui-même
                a = type(teamL[k])(teamL[k].strategy, teamL[k].stratname, 1, 1)
                match(n, teamL[k], a) #et on les fait s'affronter
                teamL[k].gain_list[l] = teamL[k].gain * (teamL[l].individuals)
            if k != l : #si il affronte une autre équipe
                match(n, teamL[k], teamL[l]) #on les fait s'affronter lors du tournoi
                teamL[k].gain_list[l] = teamL[k].gain * teamL[l].individuals #on ajoute le gain
                teamL[l].gain_list[k] = teamL[l].gain * teamL[k].individuals #on ajoute le gain

```

```

teamL[k].gain = 0 #on remet ensuite leurs gains de match à 0
teamL[l].gain = 0
teamL[k].gain_final = sum(teamL[k].gain_list) * teamL[k].individuals #on multiplie par le nombre d'individus

def trier_Hoare(L) : #trie la liste d'équipes en se basant sur leurs gains du tournoi
#algorithme de tri sur le principe du quicksort
#Il range la liste dans l'ordre décroissant. les meilleures se retrouvent au début
l = len(L)
if l == 0 or l == 1 :
    return L
else :
    pivot = L[-1]
    xhaut = [x for x in L[:-1] if x.gain_final >= pivot.gain_final]
    xbas = [x for x in L if x.gain_final < pivot.gain_final]
    return trier_Hoare(xbas) + [pivot] + trier_Hoare(xhaut)

def histogramme(Ltot, Players) : #prend en paramètre une liste d'objets de classe Player
"""
    histogramme des gains à la fin d'un tournoi
"""
s = len(Players)
x = [k for k in range(s)]
width = 0.4
BarName = [p.stratname for p in Players]
plt.barh(x, Ltot, width, color='blue')
plt.ylim(-1,s)
plt.xlim(min(Ltot)-100,max(Ltot)+100)
plt.ylabel('Total des Points', size = 20)
plt.title('Résultats du tournoi', size = 20)
plt.yticks(x, BarName, size = 12)
plt.show()

def total(n, players_list) : #joue un tournoi en affichant l'histogramme et le tournoi_fast(n,players_list) #joue le tournoi une fois
P = trier_Hoare(players_list)
Ltot = [p.gain_final for p in P] #récupère une liste des gains dans le même ordre que les équipes
histogramme(Ltot, P)

"""
    FIN DILEMME ITERE DU PRISONNIER """

```

```

"""
    EVOLUTION """
"""

def generate_personalities(S,Snom, Nindiv, zd_parameters, p_parameters) : #zd est une liste de 4 nombres, p une liste de 4 nombres
"""
    fonction pour créer la liste des équipes, chaque membre d'équipe ayant
    cette fonction doit être utilisée aussi bien pour l'évolution que pour la génération
"""
lenS = len(S) + len(zd_parameters) + len(p_parameters)
teamL = []
for i in range(len(S)) :
    teamL.append(strategies(S[i], Snom[i], Nindiv[i], lenS))
for i in range(len(zd_parameters)) :
    teamL.append(ZD(zd_parameters[i], None, Nindiv[len(S) + i], lenS ))
for i in range(len(p_parameters)) :
    teamL.append(player(p_parameters[i], None, Nindiv[len(S)+len(zd_parameters)]))

global Ntot
Ntot = sum(Nindiv)
print('teamL created')
return teamL #liste des équipes, compactée en un individu qui jouera pour tout le monde
#zd_parameters contient des sous listes de 4 nombres définissant p1, p2, p3, p4
#n_players est le nombre de players qui vont être créés

def repopulation(teamL) :
lenS = len(teamL)
gaintot = sum([k.gain_final for k in teamL if k.individuals > 0]) #gain total
for g in teamL :
    if g.individuals > 0 :
        g.repop(gaintot) #redéfinit individuals
        g.gain_list = [0]*lenS #on remet à 0 sa liste de gains, pour ne pas avoir de décalage avec les autres équipes
    else :
        g.gain_list = [0]*lenS

def generation(niter, teamL) :
tournoi_fast(niter, [k for k in teamL if k.individuals != 0])
repopulation(teamL)

def Evolution(niter, Nindiv, Ngene, S, Snom, zd_parameters, p_parameters) : #crée une population
teamL = generate_personalities(S,Snom, Nindiv, zd_parameters, p_parameters)
demography = [] #liste qui va retenir les listes des individus de chaque tournoi
for k in range(Ngene) :
    demography.append([k.individuals for k in teamL])
    generation(niter, teamL)
    print(demography[-1])
graph(Ngene, demography, teamL)
return teamL #je l'ai rajouté car j'en ai besoin pour la discréétisation des individus

```

```

def Evolution_2(teamL, niter, Ngene) : #autre version qui prend la liste en paramètre
demography = [] #liste qui va retenir les listes des individus de chaque génération
for k in range(Ngene) :
    demography.append([k.individuals for k in teamL])
    generation(niter, teamL)
#graph(Ngene, demography, teamL)
#cette fonction est la même que celle au dessus, mais je l'ai adaptée à l'algorithme

def graph(Ngene,demography,teamL) : #trace toutes les courbes à la fin de la simulation
X = np.linspace(1,Ngene,Ngene)

style = ['-']
couleur = ['black','gray','blue','green','red','navy','aquamarine','lime','brown','purple','pink','cyan','magenta','yellow','darkblue','lightblue','darkred','darkgreen','darkcyan','darkmagenta']
s,c = len(style)-1, len(couleur)-1

for k in range(len(teamL)) :
    Y = [y[k] for y in demography]
    plt.plot(X,Y, label = teamL[k].stratname, color = couleur[rd.randint(0,c)])
plt.legend(loc = 'upper right', )
plt.xlabel('générations')
plt.ylabel('populations')
plt.show()

"""
FIN EVOLUTION """
"""

DEBUT EVOLUTION ET MUTATION DES PARAMÈTRES """

def vector_L(n_players) :
#choisit n_players vecteurs au hasard dans le cube
L = []
for k in range(n_players) :
    L.append(np.array([2*rd.random()-1, 2*rd.random()-1, 2*rd.random()]))
return L

def trier_gains(L) : #trie la liste d'équipes en se basant sur leurs gains du tournoi
#algorithme de tri sur le principe du quicksort
#Il range la liste dans l'ordre décroissant. les meilleures se retrouvent au début
l = len(L)
if l == 0 or l == 1 :
    return L
else :
    pivot = L[-1]
    i = 0
    j = l-2
    while i < j :
        if L[i][2] < pivot[2] :
            L[i], L[j] = L[j], L[i]
            j -= 1
        else :
            i += 1
    L[-1], L[i] = L[i], L[-1]
    return trier_gains(L[:i]) + [pivot] + trier_gains(L[i+1:])

```

```

xhaut = [x for x in L[:-1] if x.individuals >= pivot.individuals]
xbas = [x for x in L if x.individuals < pivot.individuals]
return trier_gains(xhaut) + [pivot] + trier_gains(xbas)

def premier_quart(teamL) : #les 25% meilleures, on leur augmente leur marqueur
    for k in teamL :
        k.marker += 1

def fusion(teamL, winners) : # fait une fusion des meilleures équipes
    l = len(teamL)
    fusions = [[rd.randint(0, l-1), rd.randint(0,l-1)] for k in range(l)] #une à deux
    for k in range(l) :
        teamL[k].strategy = (winners[fusions[k][0]].strategy + winners[fusions[k][1]].strategy)/2
        teamL[k].marker = 0 #toutes les autres, on leur remet leur marqueur à 0

def mutation(teamL, n) : #ne modifie qu'une seule coordonnée légèrement du paramètre
    for k in teamL :
        pk = rd.randint(0,2)
        k.strategy[pk] += rd.random()/(n+1)-1/(2*n+2)
        k.marker = 0

def naissance(teamL) : #on supprime les moins bonnes et on les remplace par des nouvelles
    for k in teamL :
        for l in range(3) : #on sait qu'il y a 3 paramètres par équipe, c'est un couple
            k.strategy[l] = rd.random()*2-1
        k.marker = 0

def redefinir(teamL, k) :
    lt = len(teamL)//4
    premier_quart(teamL[:lt])
    mutation(teamL[lt:2*lt], k) #on fusionne des couples créés sur le meilleur
    fusion(teamL[2*lt:3*lt], teamL[:lt])
    naissance(teamL[3*lt:])

def genetique(Nindiv, n_players, niter, Ngene, Nevol) :
    """
    Nindiv est un entier, n_players le nombre d'équipes dans la liste, niter le nombre de
    Ngene le nombre de générations du jeu d'évolution, Nevol le nombre de fois
    que la liste de stratégies subit plusieurs modifications génétiques :
    1/4 de la population suivante est constituée des meilleures de la population
    1/4 sont des fusions
    1/4 sont des mutations (des paramètres fixes et un autre modifié randomiquement)
    1/4 sont des nouvelles stratégies
    """

```

```

"""
vectorL = vector_L(n_players) #on initialise une liste de paramètres random
teamL = generate_personalities([], [], [Nindiv]*n_players, [], vectorL) #crée
for n in range(Nevol) : #on va répéter le tournoi n fois
    print(str(n*100/Nevol) + '%')
    print([k.marker for k in teamL])
    for k in teamL : #on met à 0 leurs populations a la fin de chaque tour
        k.individuals = Nindiv
    Evolution_2(teamL, niter, Ngene)
    teamL = trier_gains(teamL) #on fait un rang des équipes en fonction de
    print([k.individuals for k in teamL])
    redefinir(teamL, n) #fonction qui réalise les 4 types d'évolution génératrices
    print('100%')

```

```

#test(teamL)
fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.set_xlabel('p1')
ax.set_ylabel('p2')
ax.set_zlabel('p3')
#plt.scatter([k.strategy[1] for k in teamL if k.individuals !=0], [k.strategy[0] for k in teamL], [k.strategy[2] for k in teamL])
im = ax.scatter([k.strategy[0] for k in teamL], [k.strategy[1] for k in teamL], [k.strategy[2] for k in teamL])
fig.colorbar(im, ax=ax)
plt.show()

return teamL

```

```

def test(teamL) :
    S = [tit_for_tat, gradual, pavlov, hard_majo, soft_majo, mechante, spiteful]
    Snom = ['tit_for_tat', 'gradual', 'pavlov', 'hard_majo', 'soft_majo', 'mechante', 'spiteful']
    stratL = generate_personalities(S, Snom, [100]*(10 + len(teamL)), [], [k.strategy[0] for k in teamL], total=100,stratL)

```

"" FIN EVOLUTION ET MUTATION DES PARAMÈTRES """

"" DEBUT DISCRETISATION DES PARAMETRES """

```

def select(precision, Nteam) :
    val = np.linspace(-1, 1, precision+1) #on découpe [0,1] en précision intervalles
    l = len(val)
    L_param = []
    for k1 in range(l) : #on regarde chaque vecteur de la grille
        for k2 in range(l) :
            for k3 in range(l) :

```

```

                if bol == True :#and abs(vect[0]) + abs(vect[1]) >= abs(vect[2])
                    #if abs(vect[0]) <= abs(vect[1]) + abs(vect[2]) : #et les 2 premiers sont proches
                    L_param.append(vect) #####modification temporaire du paramètre
                param = [L_param[rd.randint(0, len(L_param)-1)] for k in range(Nteam)]
                param.append(np.array((0.2, 0.2, 0.8))) #tit_for_tat
                param.append(np.array((0.001, 1, 0))) #spiteful

                print('param defined')
                return L_param

```

```

def player_discretisation(niter, precision, Ngene, Nteam) :
    param = select(precision, Nteam)

    teamL = generate_personalities([], [], [100]*len(param), [], param)
    Evolution_2(teamL, niter, Ngene)
    fig = plt.figure()
    ax = fig.add_subplot(111, projection = '3d')
    #plt.scatter([k.strategy[1] for k in teamL if k.individuals !=0], [k.strategy[0] for k in teamL], [k.strategy[2] for k in teamL])
    im = ax.scatter([k.strategy[0] for k in teamL], [k.strategy[1] for k in teamL], [k.strategy[2] for k in teamL])
    fig.colorbar(im, ax=ax)
    plt.show()

    return teamL

```

```

def convergence_player() :
    teamL = player_discretisation(100, 20, 30, 400) #on joue une fois la discréteisation
    winner = [k for k in teamL if k.individuals > 0] #on garde que celles qui ont survécu
    win_cop = len(winner) + 1
    print('discretisation terminée')

    while win_cop != len(winner) : #tant que on arrive à en sélectionner, on joue
        win_cop = len(winner)
        print(win_cop)
        for k in winner :
            k.individuals = 100
        Evolution_2(winner, 100, 30) #on joue l'évolution pour retirer celles qui ont survécu
        winner = [k for k in winner if k.individuals > 0]

    fig = plt.figure()
    ax = fig.add_subplot(111, projection = '3d')
    ax.scatter([k.strategy[0] for k in winner], [k.strategy[1] for k in winner], [k.strategy[2] for k in winner])
    plt.show()

    return winner

```

```

def plot_plans(teamL) :
    fig = plt.figure()
    ax = fig.add_subplot(111, projection = '3d')
    ax.scatter([k.strategy[0] for k in teamL], [k.strategy[1] for k in teamL],
               [k.strategy[2] for k in teamL])

    X = np.linspace(-1, 1, 2)
    Y = np.linspace(-1, 1, 2)
    X,Y = np.meshgrid(X, Y)
    Z = 1*X/1.2 - Y/1.1
    #Y2 = X-Z
    plt.gca().plot_surface(X, Y, Z)
    #plt.gca().plot_surface(X, Y2, Z)

    plt.show()

```

```

def player_discretisation_zoom(precision) :
    val1 = np.linspace(-1, 1, precision+1)
    val2 = np.linspace(-1, 1, precision+1) #[-1,1] découpé en précision intervalle
    val3 = np.linspace(-1, 1, precision+1)
    l = len(val1)

    vector = vector_L(0)

    L_param = [] #liste de tous les paramètres possibles
    for k1 in range(l) :
        for k2 in range(l) :
            for k3 in range(l) :
                L_param.append(np.array((val1[k1], val2[k2], val3[k3])))

    teamL = Evolution(100, [100]*len(L_param+vector), 40, [], [], [], L_param + vector)
    fig = plt.figure()
    ax = fig.add_subplot(111, projection = '3d')
    #plt.scatter([k.strategy[1] for k in teamL if k.individuals !=0], [k.strategy[0] for k in teamL], [k.strategy[2] for k in teamL])
    im = ax.scatter([k.strategy[0] for k in teamL], [k.strategy[1] for k in teamL], [k.strategy[2] for k in teamL])

    fig.colorbar(im, ax=ax)
    plt.show()

    return teamL

```

```

def zd_discretisation(niter, precision, Ngene, Nteam) :

    val1 = np.linspace(0, 1, precision+1)
    val2 = np.linspace(0, 1, precision+1) #[-1,1] découpé en précision intervalle
    val3 = np.linspace(0, 1, precision+1)
    val4 = np.linspace(0, 1, precision+1)

```

```

    l = len(val1)
    L_param = [] #liste de tous les paramètres possibles
    for k1 in range(l) :
        for k2 in range(l) :
            for k3 in range(l) :
                for k4 in range(l) :
                    L_param.append(np.array((val1[k1], val2[k2], val3[k3], val4[k4])))

    ll = len(L_param)-1
    param = [L_param[rd.randint(0,ll)] for k in range(Nteam)]

    teamL = Evolution(niter, [100]*len(param), Ngene, [], [], param, [])

    return teamL

```

```

def etude_liste_zd(teamL) :
    d = trier_gains(teamL)
    E = [k for k in d if k.individuals > 0]
    F = [k for k in d if k.individuals == 0]
    e, f = len(E), len(F)
    G = []
    p1 = 0
    q1 = 0
    for k in range(e) :
        G.append(E[k].strategy[1] + E[k].strategy[2] + E[k].strategy[3])
        if E[k].strategy[0] == 1 :
            p1 += 1
    for k in range(f) :
        if F[k].strategy[0] == 1 :
            q1 += 1
    X = [k for k in range(e)]
    plt.plot(X, G, color = 'black', linewidth = 1)
    plt.xlabel('rang')
    plt.ylabel('p2+p3+p4')
    return G, p1/e, q1/f

```

\*\*\*\* FIN DISCRETISATION DES PARAMETRES\*\*\*\*

\*\*\*\* EXECUTION \*\*\*\*

```

def norme(vector) :
    return np.sqrt(vector[0]**2 + vector[1]**2 + vector[2]**2)

```

```
def winner(teamL) :
    win = [k for k in teamL if k.individuals > 0]
    param = [k.strategy for k in win]

    fig = plt.figure()
    ax = fig.add_subplot(111, projection = '3d')

    ax.set_zlim(-1, 1)
    ax.xaxis.set_ticklabels(['-1', '', '-0.5', '', '0', '', '0.5', '', '1'])
    ax.yaxis.set_ticklabels(['-1', '', '-0.5', '', '0', '', '0.5', '', '1'])
    ax.zaxis.set_ticklabels(['-1', '', '-0.5', '', '0', '', '0.5', '', '1'])

    X = np.linspace(-1, 1, 2)
    Y = np.linspace(-1, 1, 2)
    X,Y = np.meshgrid(X, Y)
    Z = 1*X - Y
    plt.gca().plot_surface(X, Y, Z, cmap = "magma_r")

    im = ax.scatter([k[0] for k in param], [k[1] for k in param], [k[2] for k in param])
    fig.colorbar(im, ax=ax)
    plt.show()

def barycentre(param) :
    l = len(param)
    xg = sum([k[0] for k in param])/l
    yg = sum([k[1] for k in param])/l
    zg = sum([k[2] for k in param])/l
    return np.array((xg, yg, zg))
```