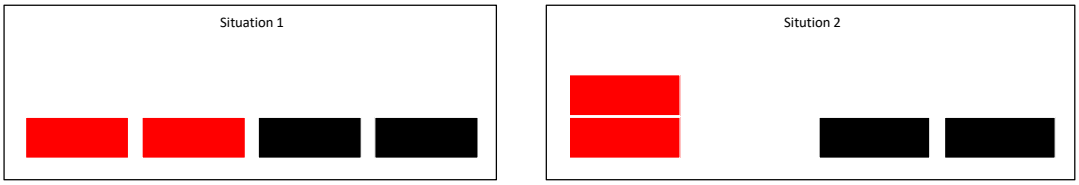
THÉORIE DES JEUX
JEUX A DEUX JOUEURS
ATTRACTEUR & STRATÉGIE

Informatique Tronc Commun

E. CLERMONT

Q1-Etat de la liste x **dans les 2 situations**



- Nombre de tablettes /couleur : N=2
- Couleurs : ci=0 pour le rouge, ci=1 pour le noir

- Situation 1: x = [[0,1] , [0,1] , [1,1] , [1,1]]
- Situation 2: x = [[0,2],[1,1],[1,1]]

2

Q2-Fonction empile(x, i, j) prenant en paramètres la liste de situations x et les indices i,j de 2 piles pi et pj tels que i≠j.

La fonction retourne la liste de listes LXij des situations atteignables par empilement.

```
def empile( x,i,j ):
    """ empile ( x : list, i : int, j : int )-> list
        entrees : x, liste de listes(ci,hi), represente la situation initiale.
                : i, j, entiers, indices des piles que l'on veut empiler
        sortie; LXij , liste de situations possibles après empilement           """
    LXij = []
    ci , hi = x[i]
    cj , hj = x[j]
    if ci==cj or hi==hj : #empilement possible
        reste = x[:i]+x[i+1:j]+x[j+1:]
        pij = [ci,hi+hj] # pile ci sur cj (arbitrairement)
        X = reste + [pij]
        X.sort()
        LXij.append(X)
        if ci!=cj: # si couleur différente => ajouter la situation avec cj au dessus
            pji = [cj,hi+hj] # pile cj sur ci
            X = reste + [pji]
            X.sort()
            LXij.append(X)
    return LXij
```

- Q3 **Fonction coups(x)** qui prend en paramètre une situation x (liste) et retourne une liste

```
def coups(x):
    """ coups(x : list) -> list
      entree: x, liste de liste(ci,hi), situation initiale
      sortie: ensemble des situations possibles avec ttes les combinaisons d'empilement possible
    """
    LX = []
    for i in range( len(x) ):
        for j in range( i+1 , len(x) ):
            LXij = empile( x,i,j )
            for X in LXij:
                if X not in LX: # On n'ajoute la situation que si elle n'a pas déjà été stockée
                    LX.append(X)
    return LX
x = [ [0,2],[0,3],[1,1],[1,2] ]
lesCoups1 = coups(x)
print( lesCoups1 )
# affiche [[[0, 5], [1, 1], [1, 2]], [[0, 3], [0, 4], [1, 1]], [[0, 3], [1, 1], [1, 4]],
[[0, 2], [0, 3], [1, 3]]]
x = [[0,1],[0,1],[1,1],[1,1]]
lesCoups2 = coups(x)
print( lesCoups2 )
# affiche [[[0, 2], [1, 1], [1, 1]], [[0, 1], [0, 2], [1, 1]], [[0, 1], [1, 1], [1, 2]],
[[0, 1], [0, 1], [1, 2]]]
```

- **Partie 2 : Création du graphe du jeu**
- Q4-**Fonction init(C, N)** prenant en argument le nombre de couleurs C et le nombre de tablettes par couleur N et renvoyant la liste x0 situation initiale du jeu triée

```
def init(C,N):
    """ init( C : int, N: int ) -> list
        entrees : C, entier positif, nombre de couleurs differentes
                : N , entier positif, nombre de tablettes par couleur
        sortie : x0, liste de listes (ci,hi) correspondant à la situation initiale
    """
    x0 = []
    for c in range(C):
        for _ in range(N):
            x0.append([c,1])
    x0.sort() # Inutile compte tenu de ma programmation avec c croissant
    return x0

test = init( 3, 4 )
print(test)
# affiche [[0, 1], [0, 1], [0, 1], [0, 1], [1, 1], [1, 1], [1, 1], [1, 1], [2, 1], [2, 1]
, [2, 1], [2, 1]]
```

5

- Q5-**Fonction récursive Tuple(L)** réalisant cette transformation de liste en Tuple.

```
def Tuple(L):
    if L==[]:
        return ()
    elif type(L[0])!=list:
        return tuple(L)
    else:
        Res = []
        for l in L:
            Res.append(Tuple(l))
        return tuple(Res)
```

6

- Q6- **Fonction graphe(C , N)** réalisant le parcours en largeur des possibilités.

```
from collections import deque
def graphe(C,N):
    """ graphe(C : int ,N:int )-> dict
        entrees:C, entier positif, nombre de couleurs differentes
               :N , entier positif, nombre de tablettes par couleur
        sortie:G, dictionnaire,le:tuple de la situation,valeur:liste de liste de situation atteignables  """
    # constitution de la situation initiale
    x0 = init(C,N)
    G = {} # initialisation du dictionnaire G qui représente le graphe
    file = deque()
    file.append( x0 ) # initialisation de la file avec la situation initiale x0
    while len( file ) != 0:
        x = file.popleft()  # recuperation du 1er element de la file
        Cle = Tuple(x)      # génération de la cle sous forme de Tuple
        LCoupsSuivants = coups(x) #Recuperation de la liste des situations atteignables depuis x
        G[Cle] = LCoupsSuivants # ajout dans le graphe du couple clé, valeur
        # ajout dans la file des differentes situations (si elles n'ont pas déjà été stockées) p
our traitement ultérieur
        for c in LCoupsSuivants:
            if Tuple(c) not in G:
                file.append(c)
    return G
```

Q7- Que représentent N1 et N2 ?

```
N1 = len(Graphe)
N2 = sum([len(Graphe[x]) for x in Graphe])
```

N1 : nombre de sommets (12 sur l'exemple)
N2 : nombre d'arêtes (16 sur l'exemple)

```
{((0, 1), (0, 1), (1, 1), (1, 1)): [[[0, 2], [1, 1], [1, 1]],
                                    [[0, 1], [0, 2], [1, 1]],
                                    [[0, 1], [1, 1], [1, 2]],
                                    [[0, 1], [0, 1], [1, 2]]],
 ((0, 2), (1, 1), (1, 1))        : [[[0, 2], [1, 2]]],
 ((0, 1), (0, 2), (1, 1))        : [[[0, 3], [1, 1]],
                                    [[0, 2], [0, 2]],
                                    [[0, 2], [1, 2]]],
 ((0, 1), (1, 1), (1, 2))        : [[[0, 2], [1, 2]],
                                    [[1, 2], [1, 2]],
                                    [[0, 1], [1, 3]]],
 ((0, 1), (0, 1), (1, 2))        : [[[0, 2],
                                     [1, 2]]],
 ((0, 2), (1, 2))                : [[[0, 4]],
                                    [[1, 4]]],
 ((0, 3), (1, 1))                : [[]],
 ((0, 2), (0, 2))                : [[[0, 4]]],
 ((1, 2), (1, 2))                : [[[1, 4]]],
 ((0, 1), (1, 3))                :  [[]],
 ((0, 4),): [],
 ((1, 4),): []
 }
```

Q7- Que représentent N1 et N2 ?

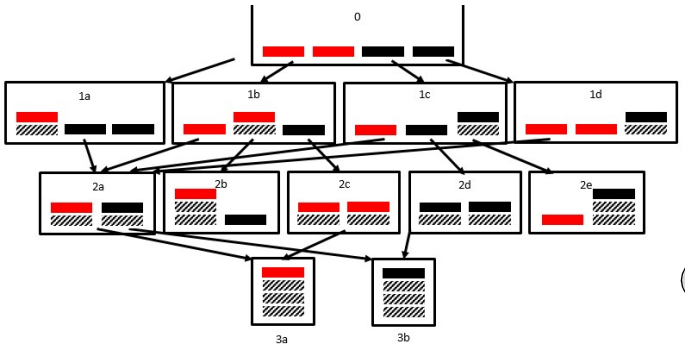
```
N1 = len(Graphe)
N2 = sum([len(Graphe[x]) for x in Graphe])
```

N1 : nombre de sommets (12 sur l'exemple)
N2 : nombre d'arêtes (16 sur l'exemple)

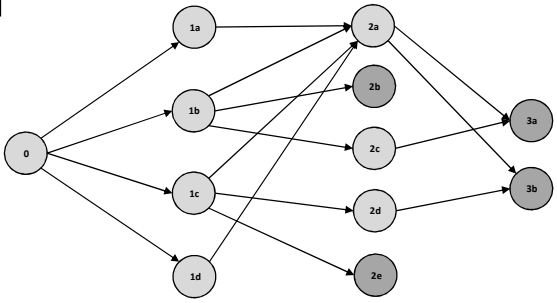
Q8- Remplissage tableau :

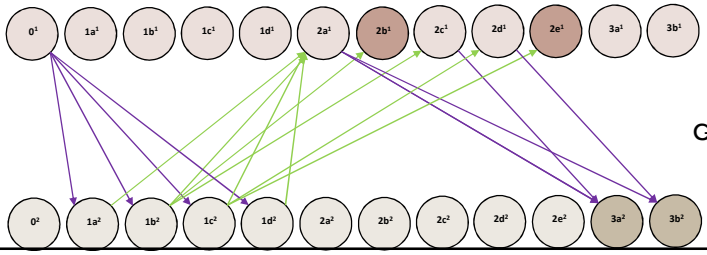
```
from time import perf_counter
C,N = 2,2 # A modifier pour chaque cas
tic = perf_counter()
Graphe = graphe(C,N)
toc = perf_counter()
T = toc - tic

print("Pour C = "+str(C)+" et N = "+str(N))
Nb_Sommets = len(Graphe) # N1
print("Sommets:",Nb_Sommets)
#N2
Nb_Aretes =sum([len(Graphe[x])for x in Graphe])
print("Arêtes:",Nb_Aretes)
print("Temps (s):",T)
```

	N =	1	2	3	4
C = 1	Sommets	1	2	3	5
	Arêtes	0	1	2	5
	Temps (s)	$2{,}1.10^{-5}$	$8{,}0.10^{-6}$	$1{,}6.10^{-5}$	$2{,}9.10^{-5}$
C = 2	Sommets	3	12	43	133
	Arêtes	2	16	90	386
	Temps (s)	$1{,}1.10^{-5}$	$8{,}8.10^{-5}$	$1{,}0.10^{-3}$	$2{,}8.10^{-2}$
C = 3	Sommets	7	92	696	4220
	Arêtes	6	234	2832	23 487
	Temps (s)	$2{,}6.10^{-5}$	$3{,}9.10^{-3}$	1,2	6560
C = 4	Sommets	23	728		
	Arêtes	48	3040		
	Temps (s)	$9{,}1.10^{-4}$	1,6		



Graphe orienté fini





Graphe biparti

- **Partie 2 : Graphe biparti**

Q9- **Fonction sommets_12(G,C,N)** prenant en argument le graphe G, N et C, et retournant les 2 Tuples des sommets S1 et S2 des joueurs J1 et J2

```
def sommets_12( G, C, N ):
    """ sommets_12( G, C, N )
        entrees : G, dictionnaire, represente le graphe
                : C, N, entiers representant le nombre de couleurs et le nombre de tablettes
        sorties: 2 tuples pour les sommets de J1 et de J2
    """
    S1 = []
    S2 = []
    n = N*C    # nombre total de tablettes = nb de piles initial
    for e in G:
        if len(e)%2 == n%2:
            S1.append(e)
        else:
            S2.append(e)
    return tuple(S1),tuple(S2) # S1 et S2 déjà Tuples, donc Tuples inutile
```

Q10- Créer les Tuples S1 et S2 dans le cas C=N=2.

```
C,N = 2,2
Graphe = graphe( C , N )
S1,S2 = sommets_12( Graphe , C , N )
```

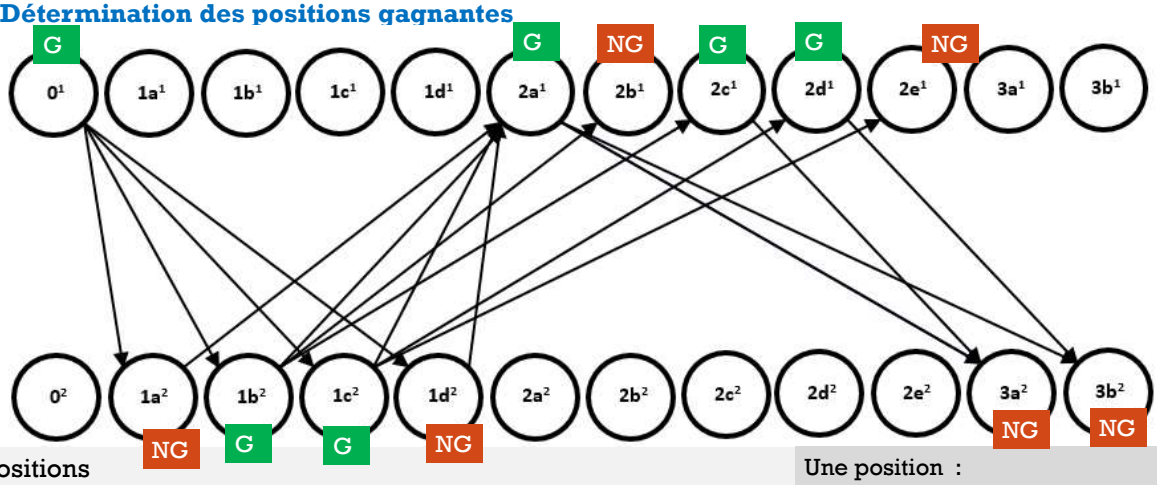
11

Q11- Pour C=N=2, et en prenant à chaque fois le 1er successeur identifié dans le graphe, **afficher une partie et préciser le joueur gagnant**

```
C,N = 2,2
Graphe = graphe(C,N) #generation du graphe du jeu
x0 = init(C,N) #situation initiale
Joueur = 1
print( "Joueur:" , Joueur )
print( "Jeu:", x0 )
lesSuccesseurs = Graphe[Tuple(x0)] #successeurs de la situation initiale

while len( lesSuccesseurs )>0:
        succ = lesSuccesseurs[0] #1er successeur
        Joueur = 3 - Joueur
        print( "Joueur:", Joueur )
        print( "Jeu:",succ )
        lesSuccesseurs = Graphe[Tuple(succ)] #successeurs du 1er successeur
```

12

Détermination des positions gagnantes



Positions
2b 2e 3a 3b pas gagnantes car pas de successeurs
2a 2c 2d gagnantes car au moins un successeur n'est pas gagnant
1a et 1d pas gagnants car tous les successeurs sont gagnants (2a)
1b gagnant car au moins un successeur pas gagnant (2b)
1c gagnant car au moins un successeur pas gagnant (2e)
0 gagnant car au moins un successeur pas gagnant (1a et 1d)
Finalement, les positions gagnantes sont: **0 1b 1c 2a 2c 2d**

Une position :
- est gagnante s'il existe au moins un successeur qui n'est pas gagnant
- **N'est pas gagnante** si :
- Elle n'a pas de successeurs
- Aucun de ses successeurs n'est pas gagnant = Tous ses successeurs sont gagnants

Q13-**Fonction est_gagnante(G,x)** prenant en argument le graphe du jeu et la position x (liste ou Tuple) et renvoyant le booléen True si la position est gagnante pour le joueur qui y joue, et False sinon

```
'''
Version 1: On vérifie que tous les successeurs sont gagnants
Pour gagner du temps, on s'arrête dès qu'un successeur pas gagnant a été trouvé
'''
def est_gagnante(G,x): # x liste ou Tuple

    lesSuccesseurs = G[Tuple(x)]

    if len(lesSuccesseurs) == 0:
        return False # Non gagnant
    else: # Tous les successeurs sont gagnants
        Tous_Gagnants = True
        for succ in lesSuccesseurs: # succ est une liste
            Tous_Gagnants = Tous_Gagnants and est_gagnante(G,succ)
            if Tous_Gagnants == False: # gagne du temps
                break  # Position gagnante
        return not(Tous_Gagnants) # Position pas gagnante
```

Q13-F**onction est_gagnante(G,x)** prenant en argument le graphe du jeu et la position x (liste ou Tuple) et renvoyant le booléen True si la position est gagnante pour le joueur qui y joue, et False sinon

```
def est_gagnante(G,x): # x liste ou Tuple

    # Détermination des successeurs de x
    lesSuccesseurs = G[Tuple(x)]

    # Aucun successeur => Pas gagnant
    if len(lesSuccesseurs) == 0:
        return False # Aucun successeur => Pas gagnant

    else: # Aucun successeur pas gagnant?
        Res = False
        for succ in lesSuccesseurs: # succ est une liste
            if not est_gagnante(G,succ): # Un pas gagnant trouvé
                Res = True # Position gagnante
                break # Gagne du temps
        return Res
```

15

- Q14- Mettre en place une **fonction dico_gagnant(G)** dont les clés sont les positions du graphe et les valeurs, le booléen True ou False indiquant si la position est gagnante ou non.

```
def dico_gagnant(G):
    dico = {}
    for x in G:
        dico[x] = est_gagnante(G,x)
    return dico

C,N = 2,2
Graphe = graphe(C,N)
x0 = init(C,N)
dico_g = dico_gagnant(Graphe)
Statut_x0 = dico_g[Tuple(x0)]
print("Le joueur 1 dispose d'une position gagnante ?",Statut_x0)

''' Résultat
Le joueur 1 dispose d'une position gagnante ? True
'''
```

16

- Q15- **Fonction dico_gagnant_opt(G)** renvoyant le dictionnaire des états gagnants des positions du graphe avec mémoïsation

```
def est_Gagnante_rec(G,x, dico): # Programmé pour que x soit un Tuple (*)
        if x in dico: # Nouveau
            return dico[x] # Nouveau
        else: # Nouveau
            if len(x) == 0:
                dico[x] = False # Nouveau
                return False
            else:
                lesSuccesseurs = G[x]
                Res = False
                for succ in lesSuccesseurs:
                    if not est_Gagnante_rec(G,Tuple(succ), dico): # (*) x est un Tuple
                        Res = True
                        break
                dico[x] = Res # Nouveau
                return Res
def dico_gagnant_opt( G ):
    dico = {}
    for x in G:
        dico[x] = est_Gagnante_rec(G,x,dico) # x est un Tuple
    return dico
```

- Q18- Fonction **init_attracteurs(G,S1)** prenant en paramètre le graphe G du jeu et le Tuple S1 des sommets du joueur J1 et renvoyant les deux dictionnaires attendus

```
def init_attracteurs(G,S1):
    """ init_attracteurs(G : dict, S1 list)
        entrees : G : dictionnaire representant le graphe
                : S1, liste des sommets gagnants du joueur1
        sortie : d1, d2, dictionnaires - cle : sommet (tuple),
                valeur : booleen inidiquant si le sommet est dans les sommets gagnants du joueur j1 """
    #initialisation des dict d1 et D2 - cle:sommet du graphe(tuple)- valeur:False par defaut
    d1 = {cle:False for cle in G}
    d2 = {cle:False for cle in G}

    for x in G: # x Tuple
        L_succ = G[Tuple(x)]
        if len(L_succ) == 0: # sans successeurs
            if x in S1: # de S1 contenant des Tuples
                d2[x] = True
            else:
                d1[x] = True
    return d1,d2
```

```
C,N = 2,2
Graphe = graphe(C,N)
S1,S2 = sommets_12(Graphe,C,N)
dA1,dA2 = init_attracteurs(Graphe,S1)

print("dA1=",dA1)
print("dA2=",dA2)
```

- Q19- **Fonction cond_1(G,di,x)** prenant en paramètre le graphe du jeu G, le dictionnaire di des attracteurs du joueur Ji et un sommet x du jeu (liste), et renvoyant le booléen True si le sommet respecte la condition (1), False sinon

```
# Q19 - cond_1
def cond_1(G,di,x):
    ''' cond_1(G : dict ,di : dict ,x : list)
        entrees : G, dictionnaire, qui représente le graphe
                : di, dictionnaires des attracteurs
                : x, liste représentant le sommet
        sortie : booléen, à True si au moins un successeur de x est à True dans di'''
    L_succ = G[Tuple(x)]
    for succ in L_succ:
        if di[Tuple(succ)] == True:
            return True
    return False
```

19

- Q20- Fonction **cond_2(G,di,x)** prenant en paramètre le graphe du jeu G, le dictionnaire di des attracteurs du joueur Ji et un sommet x du jeu (liste), et renvoyant le booléen True si le sommet possède des successeurs et respecte la condition (2), False sinon

```
def cond_2(G,di,x):
        ''' cond_2(G : dict ,di : dict ,x : list)
            entrees : G, dictionnaire, qui représente le graphe
                : di, dictionnaires des attracteurs
                : x, liste représentant le sommet
        sortie : booléen, à True si tous les successeurs de x sont True dans di'''
    L_succ = G[Tuple(x)]
    if len(L_succ) == 0:
        return False
    else:
        Res = True
        for succ in L_succ:
            Res = Res and di[Tuple(succ)]
        return Res
```

20

- Q21- **Fonction attracteurs_it(G,di,Si)** prenant en paramètre le graphe G, le dictionnaire di des attracteurs de Ji, et le Tuple Si des positions du joueur Ji, réalisant une itération de la procédure de détermination des attracteurs du joueur Ji en changeant les valeurs dans di (en place), et renvoyant True si au moins un changement (False vers True) a eu lieu, False sinon.

```
def attracteurs_it(G,di,Si):
    Changement = False
    for x in G:
        if di[x] == False:
            if x in Si:
                Cond = cond_1(G,di,x)
            else:
                Cond = cond_2(G,di,x)
            di[x] = Cond
            Changement = Changement or Cond
    return Changement
```

21

- **Q22-** Créer la **fonction attracteurs_Ji(G,di,Si)** avec les mêmes paramètres que attracteurs_it réalisant la procédure complète de création des attracteurs du joueur Ji en complétant di.

```
def attracteurs_Ji(G,di,Si):
    Changement = True
    while Changement:
        Changement = attracteurs_it(G,di,Si)
```

- **Q23-** Créer enfin **la fonction attracteurs(G,C,N)** créant et renvoyant les dictionnaires dA1 et dA2 des attracteurs des joueurs J1 et J2

```
def attracteurs(G,C,N):
    S1,S2 = sommets_12(G,C,N)
    dA1,dA2 = init_attracteurs(G,S1)
    attracteurs_Ji(G,dA1,S1)
    attracteurs_Ji(G,dA2,S2)
    return dA1,dA2

C,N = 2,2
Graphe = graphe(C,N)
dA1,dA2 = attracteurs(Graphe,C,N)
```

22

- **Q24-** Créer la **fonction dico_gagnant_att(G,C,N)** prenant en paramètres le graphe G, C et N, et renvoyant le dictionnaire des positions gagnantes

```
def dico_gagnant_att(G,C,N):
    S1,S2 = sommets_12(G,C,N)
    d1,d2 = init_attracteurs(G,S1)
    attracteurs(G,C,N)
    dico = {}
    for x in G:
        if x in S1 and d1[x]:
            dico[x] = True
        elif x in S2 and d2[x]:
            dico[x] = True
        else:
            dico[x] = False
    return dico

C,N = 2,2
Graphe = graphe(C,N)
dico_g_att = dico_gagnant_att(Graphe,C,N)
dico_g = dico_gagnant(Graphe)
test = dico_g==dico_g_att
```

23

- **Q25-** Créer la **fonction strategie_opt(G,dg,x)** prenant en paramètre le graphe G, le dictionnaire gagnant dg et une position x (liste), et renvoyant un choix de successeur respectant le choix du meilleur coup.

```
from random import randint as rd

def strategie_opt( G, dg, x ):
    L_succ = G[ Tuple(x) ]
    if len( L_succ ) == 0:
        Choix=[]
    else:
        L_Choix = []
        for succ in L_succ:
            if not dg[ Tuple(succ) ]:
                L_Choix.append(succ)
        if len(L_Choix) == 0:
            L_Choix = L_succ
        ind = rd(0,len(L_Choix)-1)
        Choix = L_Choix[ind]
    return Choix
```

24

- **Q26-** Créer la **fonction strategies_opt(G,dg)** prenant en paramètre le graphe G et le dictionnaire gagnant dg, et renvoyant un dictionnaire dico_s dont chaque clé est une position x du jeu (Tuple), et chaque valeur la solution issue du meilleur coup à jouer depuis x pour le joueur qui y est.

```
def strategies_opt(G,dg):
    dico_s = {}
    for x in G: # x Tuple
        dico_s[x] = strategie_opt(G,dg,x)
    return dico_s

C=N=2
G = graphe(C,N)
dico_g = dico_gagnant(G)
st_opt = strategies_opt(G,dico_g)
```

- **Q27-** Créer la **fonction jeu(C,N)** qui affiche le joueur disposant d'une stratégie gagnante au départ, les étapes d
- **Q28-** Utiliser la fon

```
def jeu(C,N):   # Q27 - Simulation d'un jeu
    G = graphe(C,N)
    x0 = init(C,N)
    dico_g = dico_gagnant(G) # pour afficher le joueur qui gagne en théorie
    Statut_x0 = dico_g[Tuple(x0)]
    if Statut_x0:
        print("Le joueur 1 dispose d'une position gagnante")
    else:
        print("Le joueur 2 dispose d'une position gagnante")
    st_opt = strategies_opt(G,dico_g)
    j = 1
    print('Départ:',x0)
    x = x0
    while len(x) > 0:
        x = st_opt[Tuple(x)]
        j = (3-j)%2
        print('Joueur:',j+1)
        print(x)
    print('a perdu')
''' A exécuter plusieurs fois # Q28 - Utilisation du jeu
jeu(3,3) # Joueur 1 gagne
jeu(2,3) # Joueur 2 gagne
jeu(3,2) # Joueur 2 gagne
'''
```

Dans tous les cas, le joueur disposant d'une position gagnante au départ gagne le jeu

- **Minimax et heuristique**
- **Q29-** Mettre en place la **fonction h(x,bool)** prenant en paramètre une position x du jeu (liste) et le booléen bool valant True si le joueur joue, False si c'est son adversaire, et renvoyant le résultat de l'heuristique proposée.

```
def h(x,bool): # bool = True si le joueur joue, False si adversaire
    """ h(x : list,bool: bool) -> int
        entrees : x, list, correspond à la configuration/position
                : bool, vaut True si le joueur joue, False sinon
        sortie : retourne un entier qui vaut 0,1, ou -1
    """
    Lc = coups(x) # determination de la liste des positions accessibles
    #cas ou x ne possede pas de successeur
    if len(Lc) == 0:
        if bool: # Le joueur a perdu
            return -1
        else: # L'adversaire a perdu, donc le joueur a gagné
            return 1
    else:
        return 0
test = h(Pos_0,True)
print(test)
test = h(Pos_1a,False)
print(test)
test = h(Pos_2b,True)
```

27

- **Q30-** Créer la **fonction min_max(x,p,bool)** prenant en paramètre une position x (liste), une profondeur p (entier) et le booléen représentant si c'est le coup du joueur (True) ou de l'adversaire (False) et renvoyant la valeur min-max attendue.

```
def min_max(x,p,bool):
    """ min_max(x: list, p: int , bool: bool ) -> int
        entrees : x, list, correspond à la configuration/position
                : p, int, profondeur
                : bool, vaut True si le joueur joue, False sinon
        sortie : entier qui correspond à la valeur min-max attendue   """
    Lc = coups(x)
    if len(Lc)==0 or p==0:
        return h(x,bool)
    else:
        Lh = []
        if bool: # Coup du joueur : on cherche le maximum
            for c in Lc:
                valMinMax = min_max(c,p-1,False)
                Lh.append(valMinMax)
            return max(Lh)
        else: # Coup de l'adversaire : on cherche le minimum
            for c in Lc:
                valMinMax = min_max(c,p-1,True)
                Lh.append(valMinMax)
            return min(Lh)
```

28

- **Q32-** Si vous avez du temps, proposer une **fonction min_max_opt(x,p,bool)** réalisant le même travail que min_max avec mémoïsation, observer le gain de temps pour C=N=3 et remplir le tableau des positions gagnantes au départ pour Cmax=4 et Nmax=4.

```
def min_max_opt(x,p,bool):
    def rec(x,p,bool): # x liste ou tuple
        if Tuple(x) in dico:
            return dico[Tuple(x)]
        else:
            Lc = coups(x)
            if len(Lc)==0 or p==0:
                res = h(x,bool)
                dico[Tuple(x)] = res
                return res
            else:
                Lh = []
                if bool: # Coup du joueur
                    for c in Lc:
                        Min_max = rec(c,p-1,False)
                        Lh.append(Min_max)
                    res = max(Lh)
                    dico[Tuple(x)] = res
                    return res
                else: # Coup de l'adversaire
                    for c in Lc:
                        Min_max = rec(c,p-1,True)
                        Lh.append(Min_max)
                    res = min(Lh)
                    dico[Tuple(x)] = res
                    return res
    dico = {}
    return rec(x,p,bool)
```

- **Q33-** Créer la **fonction choix_ind_max(L)** prenant en paramètre une liste L et renvoyant aléatoirement l'un des indices python des maximums de L

```
def choix_ind_max( L ):
    """ choix_ind_max( L: list ) -> int
        entree : L, liste de valeurs,
        sortie : Ind, entier qui correspond à l'indice d'une des valeurs maximales
            (s'il y en a plusieurs, l'indice sera choisi aleatoirement)
    """
    maxi = max(L)  #determination de la valeur maximale
    L_ind = []   #L_ind, liste qui sert a stocker les indices des valeurs maximales
    for i in range(len(L)):
        if L[i] == maxi:
            L_ind.append(i)
    i = rd(0,len(L_ind)-1)
    Ind = L_ind[i]
    return Ind
```

- **Q34-** Créer la **fonction strategie_h(x,p)** renvoyant le meilleur choix de coup depuis x avec une étude à la profondeur p.

```
def strategie_h(x,p):
    """ strategie_h(x,p) -> int
        entrees : x,liste position/configuration
                : p, int, profondeur
        sortie : Choix, liste qui correspond à la position suivante
    """
    Lc = coups(x)
    if len(Lc) == 0:
        Choix=[]
    else:
        if p==0: # Choix aléatoire
            ind = rd(0,len(Lc)-1)
        else:
            Lm = []
            for c in Lc:
                valMinMax = min_max(c,p-1,False) # Penser à mettre False
                Lm.append(valMinMax)
            ind = choix_ind_max(Lm)
        Choix = Lc[ind]
    return Choix
```

31

- **Q35-** Créer la **fonction jeu_h(C,N,p)** simulant un jeu pour les valeurs de C et N avec une étude à chaque coup à la profondeur p.

```
def jeu_h(C,N,p):
    """ jeu_h(C:int, N:int , p:int)
        entrees : C,N, entiers, nombre de couleurs et nombre de jetons
                : p,, entier, profondeur
    """
    x0 = init(C,N)
    j = 1
    print('Départ:',x0)
    x = x0
    while len(x) > 0:
        x = strategie_h(x,p)
        j = (3-j)%2
        print('Joueur:',j+1)
        print(x)
    print('a perdu')
```

- **Q36-** Utiliser la fonction jeu_h pour différentes situations et observer les résultats.

```
# Q36 - Utilisation du jeu

jeu_h(2,2,2) # joueur 1 gagnant
```

32