Recherche de stratégies efficaces pour le jeu Hanamikoji

Code du TIPE

Table des matières

1 Algotihme utilisant des astuces `champion_astuce.py`

1.1 Fonctions de bases / Simulations

```python
from api import *
import time
from math import inf

def valeur(g, possession=False):
    """
    Renvoie la valeur associé au numéro de la carte objet g
    Le paramètre possession, s'il est égal à True,
    renvoie une valeur de 0 il est impossible de changer qui à la possession de la geicha
    """
    if possession and possede_abs(g) != 0:
        return 0
    elif g in [0, 1, 2]:
        return 2
    elif g in [3, 4]:
        return 3
    elif g == 5:
        return 4
    elif g == 6:
        return 5
    else:
        return 0

def nouvelle_manche():
    """
    Si une nouvelle manche commence, réinitaliser les variables avec celle de la nouvelle manche
    Renvoie True si c'est une nouvelle manche
    """
    global nb_manche, cartes, sec, defausse
    if manche() != nb_manche:
        print("C'est une nouvelle manche")
        l_cartes = cartes_en_main()
        sec = -1
        defausse = [0 for _ in range(7)]
        nb_manche = manche()
        cartes = [0 for _ in range(7)]
        for c in l_cartes:
            cartes[c] += 1
        return True
    else:
        return False

def nb_validee(j, v=False):
    """
    Compte le nombre total de cartes que le joueur j a déjà validé
```

```
    Le paramètre v, s'il est a True, compte une carte de plus si le joueur a déjà joué l'action validé
    """
    t = 0
    for i in range(7):
        t += nb_cartes_validees(j, i)
    if est_jouee_action(j, action.VALIDER) and v:
        t += 1
    return t

def possede_abs(
    g,
    s=True,
    defausse=[0, 0, 0, 0, 0, 0, 0],
    add_m=[0, 0, 0, 0, 0, 0, 0],
    add_a=[0, 0, 0, 0, 0, 0, 0],
):
    """
    Paramètres d'entrées :
    g -> la geicha à tester
    s -> si True, prend en compte la carte secrète si on en a validé une
    defausse -> la défausse à prendre en compte
    add_m -> les cartes que l'on rajoute à soi-même pour faire une simulation
    add_a -> les cartes que l'on rajoute à l'adversaire pour faire une simulation

    Sortie : Renvoie qui va posseder la geicha à la fin de la manche :
    1 -> si c'est moi qui la possède
    -1 -> si c'est l'adversaire
    0 -> si le résultat n'est pas encore définit

    """
    global MOI, ADV, sec
    nb_cartes_jeu = valeur(g) - defausse[g]
    cartes_v_m = nb_cartes_validees(MOI, g) + add_m[g] #Mes cartes validées
    cartes_v_a = nb_cartes_validees(ADV, g) + add_a[g] #Les cartes validées par l'adversaires visibles
    nb_r_m = 8 - nb_validee(MOI, True)#Le nombre de cartes que je peux encore valider
    nb_r_a = 8 - nb_validee(ADV)#Le nombre de cartes que l'adversaire peut encore valider (+ sa carte secrète)
    for i in add_m:
        nb_r_m -= i
    for i in add_a:
        nb_r_a -= i
    diff = cartes_v_m - cartes_v_a
    if s and sec == g:
        cartes_v_m += 1
    majorite = [0, 0] #Le nombre de cartes qu'il faut avoir pour obtenir la majorité absolue
    if possession_geisha(g) == joueur.EGALITE:
```



```
        if nb_cartes_jeu in [0, 1]:
            majorite = [1, 1]
        elif nb_cartes_jeu in [2, 3]:
            majorite = [2, 2]
        else:
            majorite = [3, 3]
    else:
        tab_maj = [[0, 1],[1, 1],[1, 2],[2, 2],[2, 3],[3, 3]]
        majorite = tab_maj[nb_cartes_jeu]
        if possession_geisha(g) == ADV:#On inverse si c'est l'adversaire qui a la possession
            majorite[0], majorite[1] = majorite[1], majorite[0]
    if cartes_v_m >= majorite[0]:
        return 1
    elif cartes_v_a >= majorite[1]:
        return -1
    elif nb_r_m + diff < 0 or (nb_r_m + diff == 0 and possession_geisha(g) == ADV):#Si je ne peux pas placer assez de cartes pour le dépasser
        print("Special adv")
        return -1
    elif nb_r_a - diff < 0 or (nb_r_a - diff == 0 and possession_geisha(g) == MOI):#S'il ne peut pas placer assez de cartes pour me dépasser
        print("Special moi")
        return 1
    else:
        return 0


def possede_relatif(
    g,
    s=True,
    defausse=[0, 0, 0, 0, 0, 0, 0],
    add_m=[0, 0, 0, 0, 0, 0, 0],
    add_a=[0, 0, 0, 0, 0, 0, 0],
):
    """
    Paramètres d'entrées :
    g -> la geicha à tester
    s -> si True, prend en compte la carte secrète si on en a validé une
    defausse -> la défausse à prendre en compte
    add_m -> les cartes que l'on rajoute à soi-même pour faire une simulation
    add_a -> les cartes que l'on rajoute à l'adversaire pour faire une simulation

    Sortie : Renvoie qui possède la geicha actuellement :
    1 -> si c'est moi qui la possède
    -1 -> si c'est l'adversaire
    0 -> si il y a égalité

    """
    global MOI, ADV, sec
```

```
    if (possede_abs(g,s,defausse,add_m,add_a) != 0):#Si c'est vrai absolument, on ne regarde même pas relativement
        return possede_abs(g,s,defausse,add_m,add_a)
    cartes_v_m = nb_cartes_validees(MOI, g) + add_m[g]
    cartes_v_a = nb_cartes_validees(ADV, g) + add_a[g]
    if s and sec == g:
        cartes_v_m += 1
    if cartes_v_m > cartes_v_a or (
        cartes_v_m == cartes_v_a and possession_geisha(g) == MOI
    ):
        return 1
    elif cartes_v_m < cartes_v_a or (
        cartes_v_m == cartes_v_a and possession_geisha(g) == ADV
    ):
        return -1
    else:
        return 0

def simul_points(
    s=True,
    de=True,
    add_m=[0, 0, 0, 0, 0, 0, 0],
    add_a=[0, 0, 0, 0, 0, 0, 0],
    defau=[0, 0, 0, 0, 0, 0, 0],
    relatif=False,
):
    """
    Paramètres d'entrées :
    s -> si True, prend en compte la carte secrète si on en a validé une
    de -> si True, prend en compte la defausse
    add_m -> les cartes que l'on rajoute à soi-même pour faire une simulation
    add_a -> les cartes que l'on rajoute à l'adversaire pour faire une simulation
    defau -> la defausse que l'on rajoute pour faire une simulation
    relatif -> si vrai, fait la simulation avec la fonction possede_relatif

    Sortie : Un tableau contenant dans la première case le score que l'on a et dans la seconde le score qu'a l'adversaire après simulation
    """
    score = [0, 0]
    if de:
        global defausse
    else:
        defausse = defau
    for i in range(7):
        if relatif:
            a = possede_relatif(i, s, defausse, add_m, add_a)
        else:
            a = possede_abs(i, s, defausse, add_m, add_a)
        if a == 1:
            score[0] += valeur(i)
```



```
        elif a == -1:
            score[1] += valeur(i)
    if score[0] >= 11 and not relatif: #Boost de points si on est sûr de gagner
        score[0] = 10000
    elif score[1] >= 11 and not relatif: #Malus de points si on est sûr de perdre
        score[1] = 1000
    return score
```

1.2 Fonction principale `jouer_tour`

Initialisation

```
MOI = 0
ADV = 0
sec = -1 #La carte secrete que l'on valide
defausse = [0 for _ in range(7)]
cartes = [0 for _ in range(7)]
nb_manche = -1

# Fonction appelee au debut du jeu
def init_jeu():
    global MOI
    global ADV
    MOI = id_joueur()
    ADV = id_adversaire()
    print(MOI, "debut jeu")

# Fonction appelée au debut du tour
def jouer_tour():
    print("C'est mon tour")
    t1 = time.time()
    global cartes, nb_manche, sec, defausse
    l_cartes = cartes_en_main()
    l_cartes.sort(reverse=True)#Tri des cartes en commencant par la plus forte
    action_non_faite = True

    if not(nouvelle_manche()):
        p = carte_piochee()
        cartes[p] += 1
```

1.2.1 Astuces

Choix trois

```
    #Vérifie si on a 3 cartes identiques
    if not est_jouee_action(MOI, action.CHOIX_TROIS):
        for i in range(len(cartes)):
            add = [[0 for _ in range(7)] for _ in range(2)]
            add[0][i] += 2
            add[1][i] += 1
```

```
            if (cartes[i] >= 3):
                cartes[i] -= 3
                e = action_choix_trois(i, i, i)
                print("Triple choix !!")
                action_non_faite = False
                break
```

Choix paquets

```
    #Si on a deux paquets identiques
    if not est_jouee_action(MOI, action.CHOIX_PAQUETS) and action_non_faite:
        t = []
        for i in range(len(cartes)):
            if cartes[i] >= 2 and possession_geisha(i) != ADV:
                t.append(i)
        while len(t) >= 2 and action_non_faite:
            add = [[0 for _ in range(7)] for _ in range(2)]
            add[0][t[0]] += 1
            add[0][t[1]] += 1
            add[1][t[0]] += 1
            add[1][t[1]] += 1
            if possede_abs(t[0], add_m=add[0], add_a=add[1]) == -1:#Vérifie
            ↪  que ça ne fait pas gagner des points à l'adversaire
                t.pop(0)
            elif possede_abs(t[1], add_m=add[0], add_a=add[1]) == -1:
                t.pop(1)
            else:
                cartes[t[0]] -= 2
                cartes[t[1]] -= 2
                e = action_choix_paquets(t[0], t[1], t[0], t[1])
                print("Deux paquets identiques !")
                action_non_faite = False
```

Defausser

```
    #Défausse idéale
    if not est_jouee_action(MOI, action.DEFAUSSER) and action_non_faite:
        for i in range(len(l_cartes)):#Pour toutes les permutations de cartes
        ↪  possibles
            for j in range(i + 1, len(l_cartes)):
                if action_non_faite:
                    add = [0, 0, 0, 0, 0, 0, 0]
                    add[l_cartes[i]] += 1
                    add[l_cartes[j]] += 1
                    if (
                        possede_abs(l_cartes[i], defausse=add) == 1
                        and possede_abs(l_cartes[j], defausse=add) == 1
                        and possede_abs(l_cartes[i]) != 1
                        and possede_abs(l_cartes[i]) != 1
                        and l_cartes[i] != l_cartes[j]
                    ):#Si on prend la possession des 2 geichas
                        cartes[l_cartes[i]] -= 1
                        cartes[l_cartes[j]] -= 1
```



```
                        e = action_defausser(l_cartes[i], l_cartes[j])
                        print("Defausse tres rentable")
                        action_non_faite = False
                        defausse = add
                        break
            if not action_non_faite:
                break
```

Valider

```
    #Action valider
    if not est_jouee_action(MOI, action.VALIDER) and action_non_faite:
        non = set()
        continuer = True
        while len(non) < 7 and action_non_faite and continuer:
            lv = [-1]
            for i in range(len(cartes) - 1, -1, -1):
                #On regarde toutes ls cartes de mêmes valeurs que l'on
                ↪  possède, que l'on a pas éliminé
                #   et dont la possession est relative
                if (
                    cartes[i] != 0
                    and possede_abs(i) == 0
                    and valeur(i) > valeur(lv[0])
                    and not (i in non)
                ):
                    lv = [i]
                elif (
                    cartes[i] != 0
                    and possede_abs(i) == 0
                    and valeur(i) == valeur(lv[0])
                    and not (i in non)
                ):
                    lv.append(i)

            if len(lv) == 1 and lv[0] != -1:
                sec = lv[0]
                if possession_geisha(lv[0]) != ADV or possede_abs(lv[0]) == 1:
                    #Si elle n'appartient pas à l'adversaire ou que on aura
                    ↪  l'avantage après
                    cartes[lv[0]] -= 1
                    e = action_valider(lv[0])
                    print("Je valide !", lv[0])
                    action_non_faite = False
                else:
                    non.add(lv[0])
                    sec = -1
            elif len(lv) > 1: #S'il y en a plusieurs
                for c in lv:
                    if (
                        possession_geisha(c) == MOI
                    ):#On valide en priorité les cartes dont on a l'avantage
                        cartes[c] -= 1
```

```
                        sec = c
                        e = action_valider(c)
                        print("Je valide !", c)
                        action_non_faite = False
                        break
                if action_non_faite:
                    for c in lv:
                        if possession_geisha(c) == joueur.EGALITE:
                            #Puis les cartes dont personne n'a d'avantage
                            cartes[c] -= 1
                            sec = c
                            e = action_valider(c)
                            print("Je valide !", c)
                            action_non_faite = False
                            break
                if action_non_faite:
                    for c in lv:
                        sec = c
                        if possede_abs(c) == 1:
                            #Puis si c'est l'adversaire qui a l'avantage et
                            ↪ que on est pas sûr de gagner la carte après
                            ↪ validation
                            #   On préfereras refaire un autre tour
                            cartes[c] -= 1
                            e = action_valider(lv[0])
                            print("Je valide !", lv[0])
                            action_non_faite = False
                            break
                        else:
                            sec = -1
                            non.add(c)
            else:
                continuer = False
            if action_non_faite:
                print("Et c'est parti pour un autre tour")
        if action_non_faite:#Si toutes les cartes ne semblent pas
        ↪ 'rentables', on valide la plus forte
            for i in range(len(cartes) - 1, -1, -1):
                if cartes[i] != 0:
                    cartes[i] -= 1
                    sec = i
                    e = action_valider(i)
                    action_non_faite = False
                    break
```

1.2.2 Simulation dans le cas échéant

Choix trois

```
    # Fait le choix triple en fonction d'un algo min-max partiel
    if not est_jouee_action(MOI, action.CHOIX_TROIS) and action_non_faite:
        choix_f = []
        maxi = -inf
        for i in range(len(l_cartes)):
            for j in range(i + 1, len(l_cartes)):
                for l in range(j + 1, len(l_cartes)):#Pour toutes les
                ↪ permutiations possibles
                    mini = inf
                    for m in range(3):#Pour chaqu'un des placements des
                    ↪ cartes
                        #Ajouts des cartes pour simulation
                        add_m = [0 for _ in range(7)]
                        add_a = [0 for _ in range(7)]
                        if m == 0:
                            add_m[l_cartes[i]] += 1
                            add_m[l_cartes[j]] += 1
                            add_a[l_cartes[l]] += 1
                        elif m == 1:
                            add_m[l_cartes[i]] += 1
                            add_a[l_cartes[j]] += 1
                            add_m[l_cartes[l]] += 1
                        else:
                            add_a[l_cartes[i]] += 1
                            add_m[l_cartes[j]] += 1
                            add_m[l_cartes[l]] += 1

                        res = simul_points(add_m=add_m, add_a=add_a,
                        ↪ relatif=True)
                        if res[0] - res[1] < mini:#On regarde la pire
                        ↪ différence
                            mini = res[0] - res[1]
                    if mini > maxi:#On regarde le meilleur choix parmis
                    ↪ toutes les simulations
                        choix_f = [l_cartes[i], l_cartes[j], l_cartes[l]]
                        score = res
                        maxi = mini

        print("Choix trois par simulation :", score)
        for c in choix_f:
            cartes[c] -= 1
        e = action_choix_trois(choix_f[0], choix_f[1], choix_f[2])
        action_non_faite = False
```

Défausser

```
    #Action defausser 2
    if not est_jouee_action(MOI, action.DEFAUSSER) and action_non_faite:
        interessante = -1
        for i in range(len(l_cartes)):
            for j in range(i + 1, len(l_cartes)):#Pour toutes les
            ↪ permutations possibles
                if action_non_faite:
                    add = [0, 0, 0, 0, 0, 0, 0]
                    add[l_cartes[i]] += 1
                    add[l_cartes[j]] += 1
                    if (
```

```
                        possede_abs(l_cartes[i], defausse=add) == 1
                        and possede_abs(l_cartes[j], defausse=add) == 1
                    ):#Defausse deux cartes identiques ou ininterressantes
                        cartes[l_cartes[i]] -= 1
                        cartes[l_cartes[j]] -= 1
                        e = action_defausser(l_cartes[i], l_cartes[j])
                        action_non_faite = False
                        defausse = add
                        break
                    elif (
                        possede_abs(l_cartes[j], defausse=add) == 1
                        and possede_abs(l_cartes[j]) != -1
                        and l_cartes[j] > interessante
                    ):#On stocke la carte qui nous semble interessante
                        interessante = l_cartes[j]
                    elif (
                        possede_abs(l_cartes[i], defausse=add) == 1
                        and possede_abs(l_cartes[i]) != -1
                        and l_cartes[i] > interessante
                    ):
                        interessante = l_cartes[i]
            if not action_non_faite:
                break
        if action_non_faite and interessante != -1:#Si on en a trouvé une interssante, on la valide avec une autre aléatoire
            l_cartes.remove(interessante)
            cartes[interessante] -= 1
            cartes[l_cartes[-1]] -= 1
            defausse[interessante] += 1
            defausse[l_cartes[-1]] += 1
            e = action_defausser(interessante, l_cartes[-1])
            print("Defausse a demi interessante")
            action_non_faite = False

    #Defausse 3
    if not est_jouee_action(MOI, action.DEFAUSSER) and action_non_faite:
        choix = []
        diff = -inf
        for i in range(len(l_cartes)):
            for j in range(i + 1, len(l_cartes)):#Toutes les permutations
                add = [0, 0, 0, 0, 0, 0, 0]
                add[l_cartes[i]] += 1
                add[l_cartes[j]] += 1
                res = simul_points(de=False, defau=add)
                if res[0] - res[1] > diff:#On regarde le choix qui nous fait perdre le moins de points
                    choix = [l_cartes[i], l_cartes[j]]
                    diff = res[0] - res[1]

        for i in range(2):
            cartes[choix[i]] -= 1
            defausse[choix[i]] += 1
```



```
        e = action_defausser(choix[0], choix[1])
        action_non_faite = False
        print("Je defausse par simulation")
```

Choix paquets

```
    #Choix paquets
    if not est_jouee_action(MOI, action.CHOIX_PAQUETS) and action_non_faite:
        maximum = max(cartes)
        if maximum == 3 or maximum == 4:#On n'a pas le choix
            for i in range(4):
                cartes[l_cartes[i]] -= 1
            e = action_choix_paquets(l_cartes[0], l_cartes[1], l_cartes[2], l_cartes[3])
            action_non_faite = False
            print("Dernier choix de paquets force (sans reel choix)")
        elif maximum == 2:# 2 cartes identiques
            num = -1
            for l in range(len(cartes)):
                if cartes[l] == 2:
                    num = l

            liste_d = l_cartes.copy()
            add = [[[0 for _ in range(7)] for _ in range(2)] for _ in range(2)]#Liste de toutes les possiblités
            liste_d.remove(num)
            liste_d.remove(num)
            #Remplissage de la liste
            add[1][0][num] += 2
            add[1][1][liste_d[0]] += 1
            add[1][1][liste_d[1]] += 1
            add[0][0][num] += 1
            add[0][1][num] += 1
            add[0][0][liste_d[0]] += 1
            add[0][1][liste_d[1]] += 1


            if possede_abs(
                num, defausse=defausse, add_m=add[0][0], add_a=add[0][1]
            ) == -1 and possede_abs(
                num, defausse=defausse, add_m=add[1][0], add_a=add[1][1]
            ):
                e = action_choix_paquets(num, num, liste_d[0], liste_d[1])
                action_non_faite = False
                print("Choix paquets optimal avec les deux identiques du meme cote ")
            elif possede_abs(num, defausse=defausse, add_m=add[0][0], add_a=add[0][1]) == 1:
                e = action_choix_paquets(num, liste_d[0], num, liste_d[1])
                action_non_faite = False
                print("Choix paquets optimal avec les deux identiques dans des paquets differents ")
        else:
```

```
            liste_d = l_cartes.copy()
            add = [[[0 for _ in range(7)] for _ in range(2)] for _ in
            ↪ range(3)]
            #Remplissage de la liste
            add[0][0][liste_d[0]] += 1
            add[0][1][liste_d[1]] += 1
            add[0][0][liste_d[2]] += 1
            add[0][1][liste_d[3]] += 1
            add[1][0][liste_d[0]] += 1
            add[1][1][liste_d[2]] += 1
            add[1][0][liste_d[1]] += 1
            add[1][1][liste_d[3]] += 1
            add[2][0][liste_d[0]] += 1
            add[2][1][liste_d[2]] += 1
            add[2][0][liste_d[1]] += 1
            add[2][1][liste_d[3]] += 1
        #Simulation min-max partielle
        if action_non_faite:
            choix_f = []
            maxi = -inf
            for i in range(len(add)):
                mini = inf
                for j in range(2):
                    l = 1 #1 si j = 0; 0 si j = 1
                    if j == 1:
                        l = 0
                    res = simul_points(add_m=add[i][j], add_a=add[i][l])
                    if res[0] - res[1] < mini:
                        mini = res[0] - res[1]
                if mini > maxi:
                    #Ajouts des cartes
                    choix_f = []
                    for p in range(2):
                        for c in range(7):
                            for _ in range(add[i][p][c]):
                                choix_f.append(c)
                    assert len(choix_f) == 4, "Mauvais nombre de cartes"
                    maxi = mini
            e = action_choix_paquets(choix_f[0], choix_f[1], choix_f[2],
            ↪ choix_f[3])
            action_non_faite = False
            print("Choix paquets apres simulation")

    if action_non_faite:
        print("J'ai une erreur, aucune action n'a ete faite")
    elif e != error.OK:
        print("J'ai essayer de faire une action mais j'ai eu l'erreur", e)
    else:
        print("J'ai bien fait mon action en", time.time() - t1)
    print(cartes)
    print()
```


1.3 Répondre à l'action 3

```
# Fonction appelee lors du choix entre les trois cartes lors de l'action de
# l'adversaire (cf tour_precedent)
def repondre_action_choix_trois():
    nouvelle_manche()
    print("Repondre action 3")
    choix = []
    maxi = []
    tour_p = tour_precedent()
    lc = [tour_p.c1, tour_p.c2, tour_p.c3]#Liste des cartes possibles
    for i in range(3):
        maxi.append(lc[i])
        choix.append(i)
    if maxi[0] == maxi[1] and maxi[1] == maxi[2]:
        print("Trois cartes identiques")
        e = repondre_choix_trois(choix[0])
    else:
        res = []
        choix_m = 0
        diff = -inf
        for l in range(3):#Pour chaque cartes possibles
            liste_cartes = lc.copy()
            liste_cartes.remove(maxi[l])
            add_m = [0, 0, 0, 0, 0, 0, 0]
            add_a = [0, 0, 0, 0, 0, 0, 0]
            add_m[maxi[l]] += 1
            add_a[liste_cartes[0]] += 1
            add_a[liste_cartes[1]] += 1
            res.append(simul_points(add_m=add_m, add_a=add_a,
            ↪ relatif=True))#Simulations
            if res[l][0] - res[l][1] > diff:
                diff = res[l][0] - res[l][1]
                choix_m = choix[l]
        e = repondre_choix_trois(choix_m)
    print(len(choix), e)
    print()
```

1.4 Répondre à l'action 4

```
# Fonction appelee lors du choix entre deux paquets lors de l'action de
# l'adversaire (cf tour_precedent)
def repondre_action_choix_paquets():
    nouvelle_manche()
    print("Repondre paquet")
    tour_p = tour_precedent()
    lc = [tour_p.c1, tour_p.c2, tour_p.c3, tour_p.c4]#Liste des cartes
    ↪ possibles
    if (lc[0] == lc[2] and lc[1] == lc[3]) or (lc[0] == lc[3] and lc[1] ==
    ↪ lc[2]):
        print("Meme paquets !")
        e = repondre_choix_paquets(0)
```

```
    else:
        res = []
        choix_m = -1
        diff = -inf
        for l in range(0, 3, 2):#l = 0 ou 2
            liste_cartes = list(lc)
            add_m = [0, 0, 0, 0, 0, 0, 0]
            add_a = [0, 0, 0, 0, 0, 0, 0]
            add_m[liste_cartes.pop(l)] += 1#Les deux cartes donnés à
             ↪  l'adversaires (0,1 ou 2,3)
            add_m[liste_cartes.pop(l)] += 1
            add_a[liste_cartes[0]] += 1
            add_a[liste_cartes[1]] += 1
            res.append(simul_points(add_m=add_m, add_a=add_a, relatif=True))
            i = 0
            if l == 2:
                i = 1
            if res[i][0] - res[i][1] > diff:
                diff = res[i][0] - res[i][1]
                choix_m = i

        print(choix_m)
        e = repondre_choix_paquets(choix_m)
        print("Resultat simulation :", res)
    print("Erreur :", e)
    print()

# Fonction appelee a la fin du jeu
def fin_jeu():
    print("Fin jeu")
```


2 L'exploration du graphe de jeu partiel `graphe.py`

```
import itertools as it
from tqdm import tqdm
import os
from copy import deepcopy
import time

# paquets = [5, 5, 5, 3, 3, 3, 6, 6, 6, 6, 6, 0, 5, 0, 1, 2, 1, 2, 4, 4, 4]
# paquets = [2, 4, 5, 3, 2, 1, 0, 3, 6, 4, 6, 5, 1, 6, 6, 5, 4, 6, 5, 3, 0]
paquets = [0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 6, 6, 6, 6, 3, 4, 5, 5, 6]


def possiblite(p, n):
    a = it.combinations(p, n)
    s = set()
    for i in a:
        s.add(i)
    return s


def C():
    if os.path.exists("save_graphe.txt"):
        print("Récupération du fichier en cours")
        fichier = open("save_graphe.txt", "r")
        temps = int(fichier.readline().strip())
        cpt = int(fichier.readline().strip())
        cpt_fin = int(fichier.readline().strip())
        tour_boucle = int(fichier.readline().strip())
        nb_pile = int(fichier.readline().strip())
        pile = []
        for _ in range(nb_pile):
            etat = []
            for _ in range(2):
                p1 = fichier.readline().strip().split(" ")
                if p1 == [""]:
                    p1 = []
                for elem in range(len(p1)):
                    p1[elem] = int(p1[elem])
                etat.append(p1)
            for _ in range(2):
                p1 = fichier.readline().strip().split(" ")
                s1 = set()
                if p1 == [""]:
                    p1 = []
                for elem in range(len(p1)):
                    try:
                        s1.add(int(p1[elem]))
                    except:
                        print(p1)
                        exit()
                etat.append(s1)
```

```
            etat.append(int(fichier.readline().strip()))
            etat.append(int(fichier.readline().strip()))
            pile.append(etat)
        pbar = tqdm(initial=tour_boucle, total=229249440000)
        print("Récupération du fichier terminé")
    else:
        pbar = tqdm(total=229249440000)
        temps = 0
        tour_boucle = 0
        cpt = 0
        cpt_fin = 0
        pile = [
            [
                paquets[:6],
                paquets[6:12],
                set([1, 2, 3, 4]),
                set([1, 2, 3, 4]),
                12,
                1,
                set(),
            ]
        ]
    temps1 = int(time.time())
    while len(pile) != 0:
        """
        if tour_boucle % 10000000 == 0:
            print(tour_boucle, len(pile))
        """
        etat = pile.pop()
        p1 = etat[0]  # paquet 1
        p2 = etat[1]  # paquet 2
        t1 = etat[2]
        t2 = etat[3]
        tour = etat[4]
        cst = etat[5]
        if tour == 20:
            cpt_fin += 1
        elif tour % 2 == 0:
            p1.append(paquets[tour])
            assert len(p1) <= 7, (p1, tour)
            for i in t1:
                a = possiblite(p1, i)
                for p in a:
                    p1suiv = deepcopy(p1)
                    t1suiv = deepcopy(t1)
                    t1suiv.remove(i)
                    for carte in p:
                        p1suiv.remove(carte)
                    cst1 = cst
                    if i == 3:
                        cst1 *= 3
                    if i == 4:
```



```
                        cst1 *= 6
                    assert len(p1suiv) <= 7, (p1suiv, p1, tour_boucle, p)
                    pile.append(
                        [p1suiv, deepcopy(p2), t1suiv, deepcopy(t2), tour + 1,
                        ↪  cst1]
                    )
                    cpt += cst1
        else:
            p2.append(paquets[tour])
            assert len(p2) <= 7, p2
            for i in t2:
                a = possiblite(p2, i)
                for p in a:
                    p2suiv = deepcopy(p2)
                    t2suiv = deepcopy(t2)
                    t2suiv.remove(i)
                    for carte in p:
                        p2suiv.remove(carte)
                    cst2 = cst
                    if i == 3:
                        cst2 *= 3
                    if i == 4:
                        cst2 *= 6
                    assert len(p1) <= 7, p1
                    pile.append(
                        [deepcopy(p1), p2suiv, deepcopy(t1), t2suiv, tour + 1,
                        ↪  cst2]
                    )
                    cpt += cst2
        tour_boucle += 1
        pbar.update(1)
        if tour_boucle % 10000000 == 0:
            print("Sauvegarde en cours...")
            # print(pile)
            fichier = open("save_graphe.txt", "w")
            fichier.write(str(temps + int(time.time()) - temps1) + "\n")
            fichier.write(str(cpt) + "\n")
            fichier.write(str(cpt_fin) + "\n")
            fichier.write(str(tour_boucle) + "\n")
            fichier.write(str(len(pile)) + "\n")
            for etat_p in pile:
                for l in range(4):
                    for p1_c in etat_p[l]:
                        fichier.write(str(p1_c) + " ")
                    fichier.write("\n")
                fichier.write(f"{etat_p[4]}\n{etat_p[5]}\n")
            fichier.close()
            print("Sauvegarde terminée")
    pbar.close()
    print("Enregistement")
    fichier = open("fin.txt", "w")
    fichier.write(
```

```
        "Temps en secondes : " + str(temps + int(time.time()) - temps1) + "\n"
    )
    fichier.write("Nombre de noeuds dans le graphe maximal : " + str(cpt) +
    ↪  "\n")
    fichier.write("Nombre d'états finaux : " + str(cpt_fin) + "\n")
    fichier.write("Nombre de tours de boucle : " + str(tour_boucle) + "\n")
    fichier.close()
    print("Terminé !")

    return cpt


print(C())
```


3 L'algorithme regardant les états finaux

3.1 Le code de l'algorithme `champion_etats_finaux.c`

3.1.1 Structures et fonctions de bases

```c
#include "api.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "calcul_score.h"
#include <sys/time.h>
#include "convertir.h"

typedef struct etat
{
    int *valide_adv; // Les cartes déjà validés par mon adversaire
    int *valide_moi; // Les caryes déjà validés par moi
    int *avantage;   // -1 si l'avantage est à mon adversaire +1 si c'est moi
    ↪  0 sinon (en fonction de chaque couleur)
} ETAT;

typedef struct game
{
    int *cartes;
    int *restantes;
    int en_main;
    int nb_restantes;
    int valide;
    int defausse1;
    int defausee2;
    bool *act_poss;
    ETAT *etat;
} GAME; // L'état actuel du jeu

typedef struct marqueurs
{
    int k;          // la taille de pointeurs // le nombre de cartes choisis
    int n;          // le nombre de cartes dans lequel on choisit
    int *cartes;    // Les cartes dans lequel on choisit
    int *pointeurs; // Les cartes choisis
} marq;

typedef struct coup
{
    int action;
    int *cartes;
} COUP;

GAME g;
int manche_accu = -1;
joueur moi;
```

```
joueur adv;
SIX*** donnes;

// LES CONSTANTES
int valeur_couleur[7] = {2, 2, 2, 3, 3, 4, 5};                    // La valeur des couleurs = au nombre de cartes
int permu_trois[3][2] = {{1, 2}, {0, 2}, {0, 1}};                 // Les deux cartes non choisis
int permu_paquet[2][4] = {{0, 1, 2, 3}, {2, 3, 0, 1}};            // Les permutations pour deux paquets
int nb_cartes_par_action[4][2] = {{1, 0}, {0, 0}, {2, 1}, {2, 2}}; // Le nombre de cartes validés par action en fonction des joueurs

long t1;
long currenttime()
{
    /*Renvoie le temps actuel en millisecondes*/
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return tp.tv_sec * 1000 + tp.tv_usec / 1000;
}

void debug_cartes(int nb, int *ens_cartes, char *nom)
{
    /*Affiche le paquet de carte ens_cartes comprendant nb cases en affichant nom avant*/
    printf("%s : ", nom);
    for (int i = 0; i < nb; i++)
    {
        printf("%d ", ens_cartes[i]);
    }
    printf("\n");
    fflush(stdout);
}

void toutes_les_cartes(int *ens_cartes)
{
    /*Initialise le tableau ens_cartes aux valeurs contenues dans valeur_couleur (variable globale)*/
    for (int i = 0; i < 7; i++)
    {
        ens_cartes[i] = valeur_couleur[i];
    }
}

void aucune_carte(int *ens_cartes)
{
    /*Initialise le tableau ens_cartes à 0*/
    for (int i = 0; i < 7; i++)
    {
        ens_cartes[i] = 0;
    }
```



```
}

void update_cartes_valides(void)
{
    /*Met à jour les cartes validées et restantes*/
    toutes_les_cartes(g.restantes);
    g.nb_restantes = 0;
    for (int i = 0; i < 7; i++)
    {
        g.etat->valide_moi[i] = nb_cartes_validees(moi, i);
        g.etat->valide_adv[i] = nb_cartes_validees(adv, i);
        g.restantes[i] = g.restantes[i] - g.etat->valide_moi[i] - g.etat->valide_adv[i] - g.cartes[i];
        g.nb_restantes += g.restantes[i];
    }
    if (!(g.act_poss[0])) // Carte validée secretement
    {
        g.restantes[g.valide] -= 1;
        g.etat->valide_moi[g.valide] += 1;
    }
    if (!(g.act_poss[1])) // Cartes defaussés
    {
        g.restantes[g.defausse1] -= 1;
        g.restantes[g.defausee2] -= 1;
        g.nb_restantes -= 2;
    }
}

void update(bool new_c)
{
    /*Met à jour l'état du jeu au début d'un tour*/
    joueur poss;
    if (manche_accu != manche()) // Pour une nouvelle manche
    {
        g.valide = -1;
        g.en_main = 0;
        g.defausse1 = -1;
        g.defausee2 = -1;
        for (int i = 0; i < 7; i++) // On regarde les avantages de chaque couleur
        {
            poss = possession_geisha(i);
            if (poss == moi)
            {
                g.etat->avantage[i] = 1;
            }
            else if (poss == EGALITE)
            {
                g.etat->avantage[i] = 0;
            }
            else
            {
```

```
                g.etat->avantage[i] = -1;
            }
        }
        for (int i = 0; i < 4; i++) // Toutes les actions sont à nouveau disponible
        {
            g.act_poss[i] = true;
        }
        int_array lc = cartes_en_main();
        aucune_carte(g.cartes);
        for (int j = 0; j < lc.length; j++) // Mets à jour les cartes en main
        {
            g.cartes[lc.items[j]] += 1;
            g.en_main += 1;
        }
        update_cartes_valides();
        manche_accu += 1;
    }
    else
    {
        if (new_c) // Si c'est juste un nouveau tour, ajoute la carte piochée (si elle existe (en fonction de new_c))
        {
            int pioche = carte_piochee();
            g.cartes[pioche] += 1;
            g.en_main += 1;
        }
        update_cartes_valides();
    }
}

void joue_valide(int c)
{
    /*Joue l'action valider avec la carte c*/
    g.act_poss[0] = false;
    g.cartes[c] -= 1;
    g.en_main -= 1;
    g.valide = c;
    error e = action_valider(c);
    if (e == OK)
    {
        printf("Action valide carte : %d\n", c);
    }
    else
    {
        printf("!!!!!!!!!!!!!!!!!! ERREUR !!!!!!!!!!!: %d\n", e);
        printf("Action valide carte : %d\n", c);
    }
}

void joue_defausse(int d1, int d2)
{
```



```
    /*Joue l'action défausser avec les cartes d1 et d2*/
    g.act_poss[1] = false;
    g.cartes[d1] -= 1;
    g.cartes[d2] -= 1;
    g.en_main -= 2;
    g.defausse1 = d1;
    g.defausee2 = d2;
    error e = action_defausser(d1, d2);
    if (e == OK)
    {
        printf("Action defausser cartes : %d, %d\n", d1, d2);
    }
    else
    {
        printf("!!!!!!!!!!!!!!!!!! ERREUR !!!!!!!!!!!: %d\n", e);
        printf("Action defausser cartes : %d, %d\n", d1, d2);
    }
}

void joue_trois(int c1, int c2, int c3)
{
    /*Joue l'action 3 avec les cartes c1, c2 et c3*/
    g.act_poss[2] = false;
    g.cartes[c1] -= 1;
    g.cartes[c2] -= 1;
    g.cartes[c3] -= 1;
    g.en_main -= 3;
    error e = action_choix_trois(c1, c2, c3);
    if (e == OK)
    {
        printf("Action triple choix cartes : %d, %d, %d\n", c1, c2, c3);
    }
    else
    {
        printf("!!!!!!!!!!!!!!!!!! ERREUR !!!!!!!!!!!: %d\n", e);
        printf("Action triple choix cartes : %d, %d, %d\n", c1, c2, c3);
    }
}

void joue_quatre(int c11, int c12, int c21, int c22)
{
    /*Joue l'action des paquets avec les cartes c11 et c12 d'un coté, et les cartes c21 et c22 de l'autre*/
    g.act_poss[3] = false;
    g.cartes[c11] -= 1;
    g.cartes[c12] -= 1;
    g.cartes[c21] -= 1;
    g.cartes[c22] -= 1;
    g.en_main -= 4;
    error e = action_choix_paquets(c11, c12, c21, c22);
    if (e == OK)
    {
```

```
        printf("Action choix paquets cartes : %d %d %d %d\n", c11, c12, c21, c22);
    }
    else
    {
        printf("!!!!!!!!!!!!!!!!!! ERREUR !!!!!!!!!!!: %d\n", e);
        printf("Action choix paquets cartes : %d, %d, %d, %d\n", c11, c12, c21, c22);
    }
}
```

3.1.2 Simulations

```
marq *init_marqueur(int k, int n, int *cartes)
{
    /*Initialise le marqueur pour choisir k cartes parmi les n cartes dans cartes*/
    marq *m = malloc(sizeof(marq));
    m->cartes = cartes;
    m->k = k;
    m->n = n;
    if (k <= n)
    {
        m->pointeurs = malloc(k * sizeof(int));
        int point = 0;
        int carte = 0;
        while (point < k && carte < 7)
        {
            // prend la première carte possible
            for (int i = 0; i < cartes[carte] && point < k; i++)
            {
                m->pointeurs[point++] = carte;
            }
            carte++;
        }
    }
    else
    {
        debug_cartes(7, cartes, "Cartes initialisés : ");
        m->pointeurs = NULL;
    }
    return m;
}

void choix_cartes(marq *m)
{
    /*Le marqueur m propose un nouveau choix de cartes dans la case pointeur, et NULL s'il en existe plus*/
    int dernier_non_vide = m->k - 1;
    int non_plein = 6;
    int cpt;
    int continuer = true;
```



```
    // cherche la première case vide pour faire avancer un pointeur
    while (dernier_non_vide >= 0 && continuer)
    {
        cpt = m->cartes[non_plein];
        while (cpt > 0 && dernier_non_vide >= 0 && continuer)
        {
            if (m->pointeurs[dernier_non_vide] == non_plein)
            {
                cpt--;
                dernier_non_vide--;
            }
            else
            {
                continuer = false;
            }
        }
        non_plein--;
    }

    if (dernier_non_vide < 0) // il n'y a plus de cases vides
    {
        free(m->pointeurs);
        m->pointeurs = NULL;
    }
    else
    {
        non_plein = m->pointeurs[dernier_non_vide] + 1;
        while (m->cartes[non_plein] == 0)
        {
            non_plein++;
        }
        m->pointeurs[dernier_non_vide] = non_plein;
        cpt = m->cartes[m->pointeurs[dernier_non_vide]] - 1;
        dernier_non_vide++;
        for (int i = 0; i < cpt && dernier_non_vide < m->k; i++)
        { // On repositionne les pointeurs suivants
            m->pointeurs[dernier_non_vide++] = non_plein;
        }
        non_plein++;
        while (dernier_non_vide < m->k && non_plein < 7)
        { // On décale tous les autres pointeurs
            for (int i = 0; i < m->cartes[non_plein] && dernier_non_vide < m->k; i++)
            {
                m->pointeurs[dernier_non_vide++] = non_plein;
            }
            non_plein++;
        }
        if (dernier_non_vide < m->k && non_plein >= 7) // Si les derniers pointeurs n'ont plus la place
        {
            free(m->pointeurs);
```

```
            m->pointeurs = NULL;
        }
    }
}

void free_marq(marq *m)
{
    /*Déalloue la mémoire du marqueur m*/
    if (m->pointeurs != NULL)
    {
        free(m->pointeurs);
    }
    free(m);
}

bool verification(int nb_cartes, int *cartes, int nb_restantes, int *restantes, int nb_selec, int *select, ETAT *etat, bool *act_poss_simu)
{
    /*Vérifie si la distribution des cartes est possible
    Entree : cartes : les cartes en main
             restantes : les cartes que l'on a toujours pas vu
             select : les cartes que l'on voudrait valider de notre coté
             etat : l'état du jeu
             act_poss_simu : les actions encore possible (après simulation)*/
    int nb_moi_max = 0;
    int nb_moi_min = 0;
    for (int i = 0; i < 4; i++)
    {
        if (act_poss_simu[i])
        {
            nb_moi_max += nb_cartes_par_action[i][0];
        }
    }
    int tour_a = tour();
    int t; // Le nombre de cartes que l'on a pas encore vu et qui vont arriver dans notre main
    if (moi == 1 && tour_a == 0)
    {
        t = 4;
    }
    else if (tour_a == 0 || tour_a == 1 || (tour_a == 2 && moi == 1))
    {
        t = 3;
    }
    else if (tour_a == 2 || tour_a == 3 || (tour_a == 4 && moi == 1))
    {
        t = 2;
    }
    else if (tour_a == 4 || tour_a == 5 || (tour_a == 6 && moi == 1))
    {
        t = 1;
```



```
    }
    else
    {
        t = 0;
    }
    if (nb_moi_max - t > 0)
    {
        nb_moi_min = nb_moi_max - t;
    }
    int nb_moi_borne_inf = 0;
    int nb_moi_borne_sup = 0;
    int restantes_d[7];
    int cartes_d[7];
    for (int c = 0; c < 7; c++)
    {
        restantes_d[c] = restantes[c];
        cartes_d[c] = cartes[c];
    }
    for (int c = 0; c < nb_selec; c++)
    {
        restantes_d[select[c]] -= 1;
        cartes_d[select[c]] -= 1;
    }
    for (int c = 0; c < 7; c++)
    {
        nb_moi_borne_sup += cartes[c] - cartes_d[c]; // On prend en priorité les cartes qui sont dans notre main
        if (restantes_d[c] < 0)
        {
            nb_moi_borne_inf -= restantes_d[c]; // On prend en priorié les cartes qui ne sont pas dans notre main
        }
    }
    if (nb_moi_borne_inf > nb_moi_max || nb_moi_borne_sup < nb_moi_min)
    {
        // printf("Coup impossible %d %d %d %d\n", nb_moi_borne_inf, nb_moi_max, nb_moi_borne_sup, nb_moi_min);
        return false;
    }
    else
    {
        return true;
    }
}

float simulation_coup(int nb_cartes, int *cartes, int nb_restantes, int *restantes, bool action_defausse, ETAT *etat, bool *act_poss_simu)
{
    /*Simule un coup et renvoie son score pour le coup simulé joué*/
    // INITIALISATION
    D_FLOAT *res = init_d_float(donnes);
    int nb_mon_cote = 8;
```

```c
    int nb_cote_adv = 8;
    int nb_total = nb_cartes + nb_restantes;
    int *total = malloc(7 * sizeof(int));
    int *total_s_moi = malloc(7 * sizeof(int));
    int *cartes_adv = malloc(7 * sizeof(int));
    int *cartes_moi = malloc(7 * sizeof(int));
    for (int i = 0; i < 7; i++)
    {
        nb_mon_cote -= etat->valide_moi[i];
        nb_cote_adv -= etat->valide_adv[i];
        total[i] = cartes[i] + restantes[i];
        total_s_moi[i] = cartes[i] + restantes[i];
        cartes_adv[i] = etat->valide_adv[i] + cartes[i] + restantes[i];
        cartes_moi[i] = etat->valide_moi[i];
    }
    marq *mon_cote = init_marqueur(nb_mon_cote, nb_total, total);
    marq *defausse;

    // Distribution des cartes
    while (mon_cote->pointeurs != NULL)
    {
        if (verification(nb_cartes, cartes, nb_restantes, restantes,
        ↪  nb_mon_cote, mon_cote->pointeurs, etat, act_poss_simu))
        {
            for (int i = 0; i < mon_cote->k; i++)
            {
                total_s_moi[mon_cote->pointeurs[i]] -= 1;
                cartes_adv[mon_cote->pointeurs[i]] -= 1;
                cartes_moi[mon_cote->pointeurs[i]] += 1;
            }
            if (!action_defausse)
            {
                defausse = init_marqueur(3, nb_total - mon_cote->k,
                ↪  total_s_moi);
            }
            else
            {
                defausse = init_marqueur(5, nb_total - mon_cote->k,
                ↪  total_s_moi);
            }
            while (defausse->pointeurs != NULL)
            {
                for (int i = 0; i < defausse->k; i++)
                {
                    cartes_adv[defausse->pointeurs[i]] -= 1;
                }
                ajout(res, cartes_moi, cartes_adv, g.etat->avantage,
                ↪  mon_cote->n, mon_cote->k, mon_cote->cartes,
                ↪  mon_cote->pointeurs, defausse->n, defausse->k,
                ↪  defausse->cartes, defausse->pointeurs); // On calcule le
                ↪  score de cette fin de partie
                for (int i = 0; i < defausse->k; i++)
                {
                    cartes_adv[defausse->pointeurs[i]] += 1;
                }
                choix_cartes(defausse);
            }

            // On remet à l'état initial
            free_marq(defausse);
            for (int i = 0; i < mon_cote->k; i++)
            {
                total_s_moi[mon_cote->pointeurs[i]] += 1;
                cartes_adv[mon_cote->pointeurs[i]] += 1;
                cartes_moi[mon_cote->pointeurs[i]] -= 1;
            }
        }
        choix_cartes(mon_cote);
    }

    // Libération de mémoire
    free_marq(mon_cote);
    free(total);
    free(total_s_moi);
    free(cartes_adv);
    free(cartes_moi);
    return total_simu(res);
}
```

3.1.3 Fonction principale `jouer_tour`

Initialisation

```c
// Fonction appelée au début du jeu
void init_jeu(void)
{
    // ALLOCATION DE MEMOIRE
    t1 = currenttime();
    printf("Debut match\n");
    g.cartes = malloc(7 * sizeof(int));
    g.etat = malloc(sizeof(ETAT));
    g.etat->valide_adv = malloc(7 * sizeof(int));
    g.etat->valide_moi = malloc(7 * sizeof(int));
    g.etat->avantage = malloc(7 * sizeof(int));
    g.act_poss = malloc(4 * sizeof(bool));
    g.restantes = malloc(7 * sizeof(int));
    moi = id_joueur();
    if (moi == 0)
    {
        adv = 1;
    }
    else
    {
        adv = 0;
    }
```

```
    donnes = creation("stats_cartes_doub.txt");
    printf("Fin de l'initialisation du tour : %ld\n\n", currenttime() - t1);
}

// Fonction appelée au début du tour
void jouer_tour(void)
{
    // INITIALISATION TOUR
    printf("Debut : manche %d  tour %d\n", manche(), tour());
    t1 = currenttime();
    update(true);
    printf("Update termine : 1: %d 2: %d 3 : %d 4: %d en main : %d\n",
    ↪  g.act_poss[0], g.act_poss[1], g.act_poss[2], g.act_poss[3],
    ↪  g.en_main);
    debug_cartes(7, g.cartes, "Mes cartes");
    debug_cartes(7, g.restantes, "Cartes restantes");
    float score_maxi = -50;
    COUP coup_maxi;
    coup_maxi.action = -1;
    coup_maxi.cartes = malloc(4 * sizeof(int));
    coup_maxi.cartes[0] = -1;
    coup_maxi.cartes[1] = -1;
    coup_maxi.cartes[2] = -1;
    coup_maxi.cartes[3] = -1;
    float res;
    marq *tour_simu;
    int *cartes_simu = malloc(7 * sizeof(int));
    ETAT *etat_simu = malloc(sizeof(ETAT));
    etat_simu->avantage = malloc(7 * sizeof(int));
    etat_simu->valide_adv = malloc(7 * sizeof(int));
    etat_simu->valide_moi = malloc(7 * sizeof(int));
    bool *act_poss_simu = malloc(4 * sizeof(int));
    for (int i = 0; i < 4; i++)
    {
        act_poss_simu[i] = g.act_poss[i];
    }
    for (int c = 0; c < 7; c++)
    {
        cartes_simu[c] = g.cartes[c];
        etat_simu->avantage[c] = g.etat->avantage[c];
        etat_simu->valide_adv[c] = g.etat->valide_adv[c];
        etat_simu->valide_moi[c] = g.etat->valide_moi[c];
    }
    printf("Fin de copie de l'etat du jeu : %ld\n", currenttime() - t1);
    fflush(stdout);
```

Valider

```
    ///// VALIDER UNE CARTE
    if (g.act_poss[0])
    {
        act_poss_simu[0] = false;
        tour_simu = init_marqueur(1, g.en_main, g.cartes);
```



```
        while (tour_simu->pointeurs != NULL)
        {
            cartes_simu[tour_simu->pointeurs[0]] -= 1;
            etat_simu->valide_moi[tour_simu->pointeurs[0]] += 1;
            res = simulation_coup(tour_simu->n - tour_simu->k, cartes_simu,
            ↪  g.nb_restantes, g.restantes, g.act_poss[1], etat_simu,
            ↪  act_poss_simu);
            if (res > score_maxi) // On regarde quel coup à un score maximal
            {
                score_maxi = res;
                coup_maxi.action = 1;
                coup_maxi.cartes[0] = tour_simu->pointeurs[0];
            }
            ////// APRES LA SIMULATION
            cartes_simu[tour_simu->pointeurs[0]] += 1;
            etat_simu->valide_moi[tour_simu->pointeurs[0]] -= 1;
            choix_cartes(tour_simu);
        }
        free_marq(tour_simu);
        act_poss_simu[0] = true;
    }
    printf("Fin simu valider\n");
```

Défausser

```
    // DEFAUSSER UNE CARTE
    if (g.act_poss[1])
    {
        act_poss_simu[1] = false;
        tour_simu = init_marqueur(2, g.en_main, g.cartes);
        while (tour_simu->pointeurs != NULL)
        {
            cartes_simu[tour_simu->pointeurs[0]] -= 1;
            cartes_simu[tour_simu->pointeurs[1]] -= 1;
            res = simulation_coup(tour_simu->n - tour_simu->k, cartes_simu,
            ↪  g.nb_restantes, g.restantes, false, etat_simu, act_poss_simu);
            if (res > score_maxi)
            {
                score_maxi = res;
                coup_maxi.action = 2;
                coup_maxi.cartes[0] = tour_simu->pointeurs[0];
                coup_maxi.cartes[1] = tour_simu->pointeurs[1];
            }
            ////// APRES LA SIMULATION
            cartes_simu[tour_simu->pointeurs[0]] += 1;
            cartes_simu[tour_simu->pointeurs[1]] += 1;
            choix_cartes(tour_simu);
        }
        free_marq(tour_simu);
        act_poss_simu[1] = true;
    }
    printf("Fin simu defausser\n");
```

Choix trois

```
    float score_mini;
    COUP coup_mini;
    coup_mini.cartes = malloc(4 * sizeof(int));
    coup_mini.cartes[0] = -1;
    coup_mini.cartes[1] = -1;
    coup_mini.cartes[2] = -1;
    coup_mini.cartes[3] = -1;
    if (g.act_poss[2])
    {
        act_poss_simu[2] = false;
        score_mini = 50;
        coup_mini.action = 3;
        tour_simu = init_marqueur(3, g.en_main, g.cartes);
        while (tour_simu->pointeurs != NULL)
        {
            cartes_simu[tour_simu->pointeurs[0]] -= 1;
            cartes_simu[tour_simu->pointeurs[1]] -= 1;
            cartes_simu[tour_simu->pointeurs[2]] -= 1;
            for (int c = 0; c < 3; c++)
            {
                /*On regarde le score en fonction de la carte qu'il va
                ↪  prendre*/
                etat_simu->valide_adv[tour_simu->pointeurs[c]] += 1;
                etat_simu->valide_moi[tour_simu->pointeurs[permu_trois[c][0]]]
                ↪  += 1;
                etat_simu->valide_moi[tour_simu->pointeurs[permu_trois[c][1]]]
                ↪  += 1;

                res = simulation_coup(tour_simu->n - tour_simu->k,
                ↪  cartes_simu, g.nb_restantes, g.restantes, g.act_poss[1],
                ↪  etat_simu, act_poss_simu);
                if (res < score_mini)
                { // On choisit le score minimisant les coups qu'il fait
                    score_mini = res;
                    coup_mini.cartes[0] = tour_simu->pointeurs[0];
                    coup_mini.cartes[1] = tour_simu->pointeurs[1];
                    coup_mini.cartes[2] = tour_simu->pointeurs[2];
                }
                etat_simu->valide_adv[tour_simu->pointeurs[c]] -= 1;
                etat_simu->valide_moi[tour_simu->pointeurs[permu_trois[c][0]]]
                ↪  -= 1;
                etat_simu->valide_moi[tour_simu->pointeurs[permu_trois[c][1]]]
                ↪  -= 1;
            }

            if (score_mini != 50 && score_mini > score_maxi)
            { // On choisit le coup qui maximise les 'pires choix possibles'
                score_maxi = score_mini;
                coup_maxi.action = coup_mini.action;
                coup_maxi.cartes[0] = coup_mini.cartes[0];
                coup_maxi.cartes[1] = coup_mini.cartes[1];
                coup_maxi.cartes[2] = coup_mini.cartes[2];
            }
            ////// On revient à l'état initial
            cartes_simu[tour_simu->pointeurs[0]] += 1;
            cartes_simu[tour_simu->pointeurs[1]] += 1;
            cartes_simu[tour_simu->pointeurs[2]] += 1;
            choix_cartes(tour_simu);
        }
        free_marq(tour_simu);
        act_poss_simu[2] = true;
    }
    printf("Fin simu choix 3\n");
```


Choix paquets

```
    // CHOIX DES PAQUETS
    int cpt, cpt_adv, place;
    if (g.act_poss[3])
    {
        act_poss_simu[3] = false;
        coup_mini.action = 4;
        int *cartes_choisis = malloc(7 * sizeof(int));
        int *ordre = malloc(4 * sizeof(int));
        tour_simu = init_marqueur(4, g.en_main, g.cartes);
        aucune_carte(cartes_choisis);
        while (tour_simu->pointeurs != NULL)
        {
            cartes_simu[tour_simu->pointeurs[0]] -= 1;
            cartes_simu[tour_simu->pointeurs[1]] -= 1;
            cartes_simu[tour_simu->pointeurs[2]] -= 1;
            cartes_simu[tour_simu->pointeurs[3]] -= 1;
            cartes_choisis[tour_simu->pointeurs[0]] += 1;
            cartes_choisis[tour_simu->pointeurs[1]] += 1;
            cartes_choisis[tour_simu->pointeurs[2]] += 1;
            cartes_choisis[tour_simu->pointeurs[3]] += 1;
            marq *prem_paquet = init_marqueur(2, 4, cartes_choisis);
            while (prem_paquet->pointeurs != NULL)
            {
                // Pour chaque possiblité d'associer les cartes 2 par 2
                cpt = 0;
                cpt_adv = 2;
                score_mini = 50;
                // POSSIBILITE 1 : il prend le premier paquet
                for (int c = 0; c < 7; c++)
                {
                    place = 0;
                    if ((c == prem_paquet->pointeurs[0]))
                    {
                        etat_simu->valide_moi[c] += 1;
                        ordre[cpt] = c;
                        cpt++;
                        place++;
                    }
```

```
            if (c == prem_paquet->pointeurs[1])
            {
                etat_simu->valide_moi[c] += 1;
                ordre[cpt] = c;
                cpt++;
                place++;
            }
            for (int tmp = 0; tmp < cartes_choisis[c] - place; tmp++)
            {
                etat_simu->valide_adv[c] += 1;
                ordre[cpt_adv] = c;
                cpt_adv++;
            }
        }
        if (!(cpt == 2 && cpt_adv == 4))
        {
            printf("BOUCLE NON CORRECTE : %d %d", cpt, cpt_adv);
            fflush(stdout);
        }
        assert(cpt == 2 && cpt_adv == 4);
        res = simulation_coup(tour_simu->n - tour_simu->k,
        ↪  cartes_simu, g.nb_restantes, g.restantes, g.act_poss[1],
        ↪  etat_simu, act_poss_simu);
        if (res < score_mini)
        {
            score_mini = res;
            coup_mini.cartes[0] = ordre[0];
            coup_mini.cartes[1] = ordre[1];
            coup_mini.cartes[2] = ordre[2];
            coup_mini.cartes[3] = ordre[3];
        }

        etat_simu->valide_moi[ordre[0]] -= 1;
        etat_simu->valide_moi[ordre[1]] -= 1;
        etat_simu->valide_adv[ordre[2]] -= 1;
        etat_simu->valide_adv[ordre[3]] -= 1;
        // DEUXIEME POSSIBILITE : il prend le deuxième paquet
        etat_simu->valide_adv[ordre[0]] += 1;
        etat_simu->valide_adv[ordre[1]] += 1;
        etat_simu->valide_moi[ordre[2]] += 1;
        etat_simu->valide_moi[ordre[3]] += 1;

        res = simulation_coup(tour_simu->n - tour_simu->k,
        ↪  cartes_simu, g.nb_restantes, g.restantes, g.act_poss[1],
        ↪  etat_simu, act_poss_simu);
        if (res < score_mini)
        {
            score_mini = res;
            coup_mini.cartes[0] = ordre[2];
            coup_mini.cartes[1] = ordre[3];
            coup_mini.cartes[2] = ordre[0];
            coup_mini.cartes[3] = ordre[1];
```



```
        }

        etat_simu->valide_adv[ordre[0]] -= 1;
        etat_simu->valide_adv[ordre[1]] -= 1;
        etat_simu->valide_moi[ordre[2]] -= 1;
        etat_simu->valide_moi[ordre[3]] -= 1;

        if (score_mini != 50 && score_mini > score_maxi)
        {
            score_maxi = score_mini;
            coup_maxi.action = coup_mini.action;
            coup_maxi.cartes[0] = coup_mini.cartes[0];
            coup_maxi.cartes[1] = coup_mini.cartes[1];
            coup_maxi.cartes[2] = coup_mini.cartes[2];
            coup_maxi.cartes[3] = coup_mini.cartes[3];
        }
        choix_cartes(prem_paquet);
    }
    free_marq(prem_paquet);

    ////// retour à l'état initial
    cartes_simu[tour_simu->pointeurs[0]] += 1;
    cartes_simu[tour_simu->pointeurs[1]] += 1;
    cartes_simu[tour_simu->pointeurs[2]] += 1;
    cartes_simu[tour_simu->pointeurs[3]] += 1;
    cartes_choisis[tour_simu->pointeurs[0]] -= 1;
    cartes_choisis[tour_simu->pointeurs[1]] -= 1;
    cartes_choisis[tour_simu->pointeurs[2]] -= 1;
    cartes_choisis[tour_simu->pointeurs[3]] -= 1;
    choix_cartes(tour_simu);
}
// désalocation de la mémoire
free_marq(tour_simu);
free(ordre);
free(cartes_choisis);
act_poss_simu[3] = true;
}
```

Choix de l'action

```
printf("FIN SIMU : %d\n", coup_maxi.action);
fflush(stdout);
// Joue l'action
if (coup_maxi.action == 1)
{
    joue_valide(coup_maxi.cartes[0]);
}
else if (coup_maxi.action == 2)
{
    joue_defausse(coup_maxi.cartes[0], coup_maxi.cartes[1]);
}
else if (coup_maxi.action == 3)
{
```

```
        joue_trois(coup_maxi.cartes[0], coup_maxi.cartes[1],
        ↪ coup_maxi.cartes[2]);
    }
    else if (coup_maxi.action == 4)
    {
        joue_quatre(coup_maxi.cartes[0], coup_maxi.cartes[1],
        ↪ coup_maxi.cartes[2], coup_maxi.cartes[3]);
    }
    else
    {
        printf("ERREUR ! AUCUNE ACTION JOUEE :\n");
    }
    // Désallocation de la mémoire
    free(coup_mini.cartes);
    free(coup_maxi.cartes);
    free(cartes_simu);
    free(etat_simu->avantage);
    free(etat_simu->valide_adv);
    free(etat_simu->valide_moi);
    free(etat_simu);
    free(act_poss_simu);
    printf("SCORE : %f\nTEMPS : %ld
    ↪ ms\n\n########################################\n\n", score_maxi,
    ↪ currenttime() - t1);
}
```

3.1.4 Répondre à l'action 3

```
// Fonction appelée lors du choix entre les trois cartes lors de l'action de
// l'adversaire (cf tour_precedent)
void repondre_action_choix_trois(void)
{
    // INITIALISATION
    printf("Repondre choix 3\n");
    t1 = currenttime();
    update(false);
    debug_cartes(7, g.cartes, "Mes cartes");
    debug_cartes(7, g.restantes, "Cartes restantes");
    action_jouee tp = tour_precedent();
    int cartes_3[3] = {tp.c1, tp.c2, tp.c3};

    float score_maxi = -50;
    int coup_max;
    float res;

    ETAT *etat_simu = malloc(sizeof(ETAT));
    int *restantes_simu = malloc(7 * sizeof(int));
    etat_simu->avantage = malloc(7 * sizeof(int));
    etat_simu->valide_adv = malloc(7 * sizeof(int));
    etat_simu->valide_moi = malloc(7 * sizeof(int));
    for (int c = 0; c < 7; c++)
    {
        restantes_simu[c] = g.restantes[c];
        etat_simu->avantage[c] = g.etat->avantage[c];
        etat_simu->valide_adv[c] = g.etat->valide_adv[c];
        etat_simu->valide_moi[c] = g.etat->valide_moi[c];
    }
    for (int c = 0; c < 3; c++)
    {
        restantes_simu[cartes_3[c]] -= 1;
    }
    debug_cartes(7, restantes_simu, "Restantes_simu");

    // On va simuler si on prend chaqu'une des possibilités
    for (int carte_choisie = 0; carte_choisie < 3; carte_choisie++)
    {
        etat_simu->valide_moi[cartes_3[carte_choisie]] += 1;
        etat_simu->valide_adv[cartes_3[permu_trois[carte_choisie][0]]] += 1;
        etat_simu->valide_adv[cartes_3[permu_trois[carte_choisie][1]]] += 1;
        res = simulation_coup(g.en_main, g.cartes, g.nb_restantes - 3,
        ↪ restantes_simu, g.act_poss[1], etat_simu, g.act_poss);

        if (res > score_maxi)
        {
            score_maxi = res;
            coup_max = carte_choisie;
        }
        etat_simu->valide_moi[cartes_3[carte_choisie]] -= 1;
        etat_simu->valide_adv[cartes_3[permu_trois[carte_choisie][0]]] -= 1;
        etat_simu->valide_adv[cartes_3[permu_trois[carte_choisie][1]]] -= 1;
    }
    // désallocation de mémoire
    free(restantes_simu);
    free(etat_simu->avantage);
    free(etat_simu->valide_adv);
    free(etat_simu->valide_moi);
    free(etat_simu);
    repondre_choix_trois(coup_max);
    printf("SCORE : %f\nTEMPS : %ld
    ↪ ms\n\n########################################\n\n", score_maxi,
    ↪ currenttime() - t1);
}
```

3.1.5 Répondre à l'action 4

```
// Fonction appelée lors du choix entre deux paquet lors de l'action de
// l'adversaire (cf tour_precedent)
void repondre_action_choix_paquets(void)
{
    // INITIALISATION
    printf("Repondre choix paquets\n");
    t1 = currenttime();
    update(false);
    debug_cartes(7, g.cartes, "Mes cartes");
```

```
    debug_cartes(7, g.restantes, "Cartes restantes");
    action_jouee tp = tour_precedent();
    int cartes_4[4] = {tp.c1, tp.c2, tp.c3, tp.c4};

    float score_maxi = -50;
    int coup_max;
    float res;

    ETAT *etat_simu = malloc(sizeof(ETAT));
    int *restantes_simu = malloc(7 * sizeof(int));
    etat_simu->avantage = malloc(7 * sizeof(int));
    etat_simu->valide_adv = malloc(7 * sizeof(int));
    etat_simu->valide_moi = malloc(7 * sizeof(int));
    for (int c = 0; c < 7; c++)
    {
        restantes_simu[c] = g.restantes[c];
        etat_simu->avantage[c] = g.etat->avantage[c];
        etat_simu->valide_adv[c] = g.etat->valide_adv[c];
        etat_simu->valide_moi[c] = g.etat->valide_moi[c];
    }
    for (int c = 0; c < 4; c++)
    {
        restantes_simu[cartes_4[c]] -= 1;
    }
    debug_cartes(7, restantes_simu, "Restantes_simu");

    // On va tester si on prend chaqu'un des deux paquets
    for (int carte_choisie = 0; carte_choisie < 2; carte_choisie++)
    {
        etat_simu->valide_moi[cartes_4[permu_paquet[carte_choisie][0]]] += 1;
        etat_simu->valide_moi[cartes_4[permu_paquet[carte_choisie][1]]] += 1;
        etat_simu->valide_adv[cartes_4[permu_paquet[carte_choisie][2]]] += 1;
        etat_simu->valide_adv[cartes_4[permu_paquet[carte_choisie][3]]] += 1;
        res = simulation_coup(g.en_main, g.cartes, g.nb_restantes - 4,
        ↪  restantes_simu, g.act_poss[1], etat_simu, g.act_poss);

        if (res > score_maxi)
        {
            score_maxi = res;
            coup_max = carte_choisie;
        }
        etat_simu->valide_moi[cartes_4[permu_paquet[carte_choisie][0]]] -= 1;
        etat_simu->valide_moi[cartes_4[permu_paquet[carte_choisie][1]]] -= 1;
        etat_simu->valide_adv[cartes_4[permu_paquet[carte_choisie][2]]] -= 1;
        etat_simu->valide_adv[cartes_4[permu_paquet[carte_choisie][3]]] -= 1;
    }
    // désalocation de la mémoire
    free(restantes_simu);
    free(etat_simu->avantage);
    free(etat_simu->valide_adv);
    free(etat_simu->valide_moi);
    free(etat_simu);
```



```
    repondre_choix_paquets(coup_max);
    printf("SCORE : %f\nTEMPS : %ld
    ↪  ms\n\n########################################\n\n", score_maxi,
    ↪  currenttime() - t1);
}

// Fonction appelée à la fin du jeu
void fin_jeu(void)
{
    // Désallocation de la mémoire
    free(g.cartes);
    free(g.etat->valide_adv);
    free(g.etat->valide_moi);
    free(g.etat->avantage);
    free(g.etat);
    free(g.act_poss);
    free(g.restantes);
    printf("Fin\n");
}
```

3.2 Calculer le score

3.2.1 L'entête `calcul_score.h`

```
#pragma once

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include "convertir.h"

typedef struct double_int
{
    int moi; // Sommme
    int adv; // ponderation
} D_INT;

typedef struct double_float
{
    float som;
    float pond;
    SIX ***stats;
} D_FLOAT;

bool verif_score_final(int *valide_moi, int *valide_adv);

D_INT score(int *valide_moi, int *valide_adv, int *avantage);

int diff_score(int *valide_moi, int *valide_adv, int *avantage);

D_FLOAT *init_d_float(SIX*** donnes);

```

```c
void ajout(D_FLOAT *simu, int *cartes_moi, int *cartes_adv, int *avantages, int n_m, int k_m, int *cartes_m, int *choix_m, int n_d, int k_d, int *cartes_d, int *choix_d);

float total_simu(D_FLOAT *simu);
```

3.2.2 Calculs avec la moyenne `calcul_score.c`

```c
#include "calcul_score.h"
#include "convertir.h"

int valeur_c[7] = {2, 2, 2, 3, 3, 4, 5};

bool verif_score_final(int *valide_moi, int *valide_adv)
{
    int cpt_moi = 0;
    int cpt_adv = 0;
    bool possible = true;
    for (int i = 0; i < 7; i++)
    {
        cpt_moi += valide_moi[i];
        cpt_adv += valide_adv[i];
        if (valide_moi[i] > valeur_c[i])
        {
            printf("TROP DE CARTES DE MEME VALEUR DE MON COTE POUR LA COULEUR %d : %d!\n", i, valide_moi[i]);
            possible = false;
        }
        if (valide_adv[i] > valeur_c[i])
        {
            printf("TROP DE CARTES DE MEME VALEUR DU COTE ADVERSE POUR LA COULEUR %d : %d!\n", i, valide_adv[i]);
            possible = false;
        }
    }
    if (cpt_moi != 8)
    {
        printf("LE COMPTE N'EST PAS BON DE MON COTE : %d \n", cpt_moi);
        possible = false;
    }
    if (cpt_adv != 8)
    {
        printf("LE COMPTE N'EST PAS BON DU COTE ADVERSE : %d \n", cpt_adv);
        possible = false;
    }
    return possible;
}

D_INT score(int *valide_moi, int *valide_adv, int *avantage)
{
    bool assertion = verif_score_final(valide_moi, valide_adv);
    if (!assertion)
    {
        printf("Moi %d %d %d %d %d %d %d\n", valide_moi[0], valide_moi[1], valide_moi[2], valide_moi[3], valide_moi[4], valide_moi[5], valide_moi[6]);
        printf("Adv %d %d %d %d %d %d %d\n", valide_adv[0], valide_adv[1], valide_adv[2], valide_adv[3], valide_adv[4], valide_adv[5], valide_adv[6]);
        fflush(stdout);
        assert(false);
    }
    D_INT res = {0, 0};
    for (int i = 0; i < 7; i++)
    {
        if (valide_moi[i] > valide_adv[i])
        {
            res.moi += valeur_c[i];
        }
        else if (valide_adv[i] > valide_moi[i])
        {
            res.adv += valeur_c[i];
        }
        else if (avantage[i] == 1)
        {
            res.moi += valeur_c[i];
        }
        else if (avantage[i] == -1)
        {
            res.adv += valeur_c[i];
        }
    }
    return res;
}

int diff_score(int *valide_moi, int *valide_adv, int *avantage)
{
    D_INT s = score(valide_moi, valide_adv, avantage);
    return s.moi - s.adv;
}

D_FLOAT *init_d_float(void)
{
    D_FLOAT *res = malloc(sizeof(D_FLOAT));
    res->pond = 0;
    res->som = 0;
    res->stats = NULL;
    return res;
}

float ponderation(void)
{
    return 1;
}
```

```c

void ajout(D_FLOAT *simu, int *cartes_moi, int *cartes_adv, int *avantages,
    int n_m, int k_m, int *cartes_m, int *choix_m, int n_d, int k_d, int
    *cartes_d, int *choix_d)
{
    int sco = diff_score(cartes_moi, cartes_adv, avantages);
    float p = ponderation();
    simu->som += sco * p;
    simu->pond += p;
}

float total_simu(D_FLOAT *simu)
{
    float res = simu->som / simu->pond;
    free(simu);
    return res;
}

int main()
{
    //            TESTS
    int *cm = malloc(7 * sizeof(int));
    int *ca = malloc(7 * sizeof(int));
    int *av = malloc(7 * sizeof(int));
    int cm_i[7] = {2, 2, 2, 2, 0, 0, 0};
    int ca_i[7] = {0, 0, 0, 1, 1, 3, 3};
    int av_i[7] = {0, 0, 0, 0, 0, 0, 0};
    for (int i = 0; i < 7; i++)
    {
        cm[i] = cm_i[i];
        ca[i] = ca_i[i];
        av[i] = av_i[i];
    }
    printf("%d\n", cm, ca, av, cm, ca);
    D_FLOAT *r = init_d_float();
    ajout(r, cm, ca, av, cm, ca);
    printf("%f\n", total_simu(r));
    free(ca);
    free(cm);
    free(av);
    return 0;
}
```

3.2.3 Calculs avec la moyenne pondérée statistique

L'entête de `convertir.h`

```c
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
```



```c
#include <assert.h>
#define N_MAX 238

struct sixuplet
{
    int s;
    int d;
    int t;
    int q4;
    int q5;
    int prob;
};

typedef struct sixuplet SIX;

int lin(int k, int s, int d, int t, int q4, int q5);

int *delin(int i);

SIX ***creation(char *nom);

int proba(SIX ***tab, int n, int k, SIX debut, SIX arrive);

void free_six(SIX ***s);
```

Contenu de `convertir.c`

```c
#include "convertir.h"

int lin(int k, int s, int d, int t, int q4, int q5)
{
    return 1920 * (k - 1) + 240 * s + 30 * d + 6 * t + 2 * q4 + q5;
}

int *delin(int i)
{
    int *res = malloc(6 * sizeof(int));
    res[5] = i % 2;
    i = i / 2;
    res[4] = i % 3;
    i = i / 3;
    res[3] = i % 5;
    i = i / 5;
    res[2] = i % 8;
    i = i / 8;
    res[1] = i % 8;
    i = i / 8;
    res[0] = i;
    return res;
}

SIX ***creation(char *nom)
{
```

```
    int j;
    SIX ***res_s = malloc(20 * sizeof(SIX **));
    assert(res_s != NULL);
    SIX base = {.s = -1, .d = -1, .t = -1, .q4 = -1, .q5 = -1, .prob = -1};
    for (int i = 0; i < 20; i++)
    {
        printf("%d\n", i);
        j = (1920 * (i + 1) + 240 * 8 + 30 * 8 + 6 * 5 + 2 * 3 + 2);
        res_s[i] = malloc(j * sizeof(SIX *));
        assert(res_s[i] != NULL);
        for (int k = 0; k < j; k++)
        {
            res_s[i][k] = malloc(N_MAX * sizeof(SIX));
            assert(res_s[i][k] != NULL);
            for (int l = 0; l < N_MAX; l++)
            {
                res_s[i][k][l] = base;
            }
        }
    }
    SIX en_place;
    FILE *fichier = fopen(nom, "r");
    int nb, nb_f, cpt, ind;
    int nb_f_max = 0;
    int nb_lecture = 1;
    int init[7];
    int res[6];
    fscanf(fichier, "%d", &nb);
    printf("%d\n", nb);
    for (int i = 0; i < nb; i++)
    {
        for (int j = 0; j < 7; j++)
        {
            fscanf(fichier, "%d", &init[j]);
            nb_lecture++;
        }
        fscanf(fichier, "%d", &nb_f);
        nb_lecture++;
        // printf("i : %d nb_f : %d < %d  nb_lect : %d\n", i, nb_f, nb_f_max,
        ↪  nb_lecture);
        if (nb_f > nb_f_max)
        {
            nb_f_max = nb_f;
        }
        ind = lin(init[1], init[2], init[3], init[4], init[5], init[6]);
        printf("Init : %d %d %d %d %d %d %d\n", init[0], init[1], init[2],
        ↪  init[3], init[4], init[5], init[6]);
        printf("Ind : %d\n", ind);
        cpt = 0;
        for (int j = 0; j < nb_f; j++)
        {
            for (int l = 0; l < 6; l++)
```



```
            {
                fscanf(fichier, "%d", &res[l]);
                nb_lecture++;
            }
            en_place.s = res[0];
            en_place.d = res[1];
            en_place.t = res[2];
            en_place.q4 = res[3];
            en_place.q5 = res[4];
            en_place.prob = res[5];
            printf("Res : %d %d %d %d %d %d %d\n", res[0], res[1], res[2],
            ↪  res[3], res[4], res[5], cpt);
            res_s[init[0] - 1][ind][cpt++] = en_place;
            printf("OK\n");
        }
    }
    return res_s;
}

int proba(SIX ***tab, int n, int k, SIX debut, SIX arrive)
{
    int ind = lin(k, debut.s, debut.d, debut.t, debut.q4, debut.q5);
    SIX val;
    for (int i = 0; i < N_MAX; i++)
    {
        val = tab[n][ind][i]; // PROBLEME ICI <- IMPOSSIBLE D'ACCEDER AU
        ↪  TABLEAU (mais pas d'erreurs renvoyés)
        assert(val.s != -1);
        if (val.s == arrive.s && val.d == arrive.d && val.t == arrive.t &&
        ↪  val.q4 == arrive.q4 && val.q5 == arrive.q5)
        {
            return val.prob;
        }
    }
    assert(false);
    return 0;
}

void free_six(SIX ***s)
{
    int j;
    for (int i = 0; i < 20; i++)
    {
        j = (1920 * (i + 1) + 240 * 8 + 30 * 8 + 6 * 5 + 2 * 3 + 2);
        for (int k = 0; k < j; k++)
        {
            free(s[i][k]);
        }
        free(s[i]);
    }
    free(s);
}
```

```c

int main()
{
    //      TESTS
    int t1 = time(NULL);
    creation("stats_cartes_doub.txt");
    // creation("stats_nb_doub.txt");
    int t2 = time(NULL);
    printf("Temps : %d\n", t2 - t1);
    return 0;
}
```

Calculs avec les statistiques `calcul_score.c`

```c
#include "calcul_score.h"
#include "convertir.h"

int valeur_c[7] = {2, 2, 2, 3, 3, 4, 5};

bool verif_score_final(int *valide_moi, int *valide_adv)
{
    int cpt_moi = 0;
    int cpt_adv = 0;
    bool possible = true;
    for (int i = 0; i < 7; i++)
    {
        cpt_moi += valide_moi[i];
        cpt_adv += valide_adv[i];
        if (valide_moi[i] > valeur_c[i])
        {
            printf("TROP DE CARTES DE MEME VALEUR DE MON COTE POUR LA COULEUR
            ↪  %d : %d!\n", i, valide_moi[i]);
            possible = false;
        }
        if (valide_adv[i] > valeur_c[i])
        {
            printf("TROP DE CARTES DE MEME VALEUR DU COTE ADVERSE POUR LA
            ↪  COULEUR %d : %d!\n", i, valide_adv[i]);
            possible = false;
        }
    }
    if (cpt_moi != 8)
    {
        printf("LE COMPTE N'EST PAS BON DE MON COTE : %d \n", cpt_moi);
        possible = false;
    }
    if (cpt_adv != 8)
    {
        printf("LE COMPTE N'EST PAS BON DU COTE ADVERSE : %d \n", cpt_adv);
        possible = false;
    }
    return possible;
}
```



```c

D_INT score(int *valide_moi, int *valide_adv, int *avantage)
{
    bool assertion = verif_score_final(valide_moi, valide_adv);
    if (!assertion)
    {
        printf("Moi %d %d %d %d %d %d %d\n", valide_moi[0], valide_moi[1],
        ↪  valide_moi[2], valide_moi[3], valide_moi[4], valide_moi[5],
        ↪  valide_moi[6]);
        printf("Adv %d %d %d %d %d %d %d\n", valide_adv[0], valide_adv[1],
        ↪  valide_adv[2], valide_adv[3], valide_adv[4], valide_adv[5],
        ↪  valide_adv[6]);
        fflush(stdout);
        assert(false);
    }
    D_INT res = {0, 0};
    for (int i = 0; i < 7; i++)
    {
        if (valide_moi[i] > valide_adv[i])
        {
            res.moi += valeur_c[i];
        }
        else if (valide_adv[i] > valide_moi[i])
        {
            res.adv += valeur_c[i];
        }
        else if (avantage[i] == 1)
        {
            res.moi += valeur_c[i];
        }
        else if (avantage[i] == -1)
        {
            res.adv += valeur_c[i];
        }
    }
    return res;
}

int diff_score(int *valide_moi, int *valide_adv, int *avantage)
{
    D_INT s = score(valide_moi, valide_adv, avantage);
    return s.moi - s.adv;
}

D_FLOAT *init_d_float(SIX*** donnes)
{
    D_FLOAT *res = malloc(sizeof(D_FLOAT));
    res->pond = 0;
    res->som = 0;
    res->stats = donnes;
    return res;
}
```

```c

SIX doublons(int *cartes)
{
    SIX res = {.s = 0, .d = 0, .t = 0, .q4 = 0, .q5 = 0, .prob = 0};
    for (int i = 0; i < 7; i++)
    {
        switch (cartes[i])
        {
        case 1:
            res.s += 1;
            break;
        case 2:
            res.d += 1;
            break;
        case 3:
            res.t += 1;
            break;
        case 4:
            res.q4 += 1;
            break;
        case 5:
            res.q5 += 1;
            break;
        default:
            break;
        }
    }
    return res;
}

float ponderation(D_FLOAT *simu, int n_m, int k_m, int *cartes_m, int
↪  *choix_m, int n_d, int k_d, int *cartes_d, int *choix_d)
{
    SIX doub_dep = doublons(cartes_m);
    SIX doub_fin = doublons(choix_m);
    int prob = proba(simu->stats, n_m, k_m, doub_dep, doub_fin);
    SIX doub_dep_d = doublons(cartes_d);
    SIX doub_fin_d = doublons(choix_d);
    int prob_d = proba(simu->stats, n_d, k_d, doub_dep_d, doub_fin_d);
    return prob + prob_d;
}

void ajout(D_FLOAT *simu, int *cartes_moi, int *cartes_adv, int *avantages,
↪  int n_m, int k_m, int *cartes_m, int *choix_m, int n_d, int k_d, int
↪  *cartes_d, int *choix_d)
{
    int sco = diff_score(cartes_moi, cartes_adv, avantages);
    float p = ponderation(simu, n_m, k_m, cartes_m, choix_m, n_d, k_d,
    ↪  cartes_d, choix_d);
    simu->som += sco * p;
    simu->pond += p;
}
```



```c

float total_simu(D_FLOAT *simu)
{
    float res = simu->som / simu->pond;
    free(simu);
    return res;
}
```