

Recherche de stratégies efficaces pour le jeu Hanamikoji

Code du TIPE

Table des matières

1	Algotihme utilisant des astuces champion_astuce.py	3
1.1	Fonctions de bases / Simulations	3
1.2	Fonction principale jouer_tour	7
1.2.1	Astuces	7
1.2.2	Simulation dans le cas échéant	11
1.3	Répondre à l'action 3	15
1.4	Répondre à l'action 4	15
2	L'exploration du graphe de jeu partiel graphe.py	17
3	L'algorithme regardant les états finaux	21
3.1	Le code de l'algorithme champion_etats_finaux.c	21
3.1.1	Structures et fonctions de bases	21
3.1.2	Simulations	27
3.1.3	Fonction principale jouer_tour	31
3.1.4	Répondre à l'action 3	39
3.1.5	Répondre à l'action 4	39
3.2	Calculer le score	41
3.2.1	L'entête calcul_score.h	41
3.2.2	Calculs avec la moyenne calcul_score.c	43
3.2.3	Calculs avec la moyenne pondérée statistique	45

1 Algotihme utilisant des astuces champion_astuce.py

1.1 Fonctions de bases / Simulations

```

1 from api import *
2 import time
3 from math import inf
4
5 def valeur(g, possession=False):
6     """
7     Renvoie la valeur associé au numéro de la carte objet g
8     Le paramètre possession, s'il est égal à True,
9     renvoie une valeur de 0 il est impossible de changer qui à la possession
10    ↪ de la geicha
11    """
12    if possession and possede_abs(g) != 0:
13        return 0
14    elif g in [0, 1, 2]:
15        return 2
16    elif g in [3, 4]:
17        return 3
18    elif g == 5:
19        return 4
20    elif g == 6:
21        return 5
22    else:
23        return 0
24
25 def nouvelle_manche():
26     """
27     Si une nouvelle manche commence, réinitialiser les variables avec celle de
28     ↪ la nouvelle manche
29     Renvoie True si c'est une nouvelle manche
30     """
31    global nb_manche, cartes, sec, defausse
32    if manche() != nb_manche:
33        print("C'est une nouvelle manche")
34        l_cartes = cartes_en_main()
35        sec = -1
36        defausse = [0 for _ in range(7)]
37        nb_manche = manche()
38        cartes = [0 for _ in range(7)]
39        for c in l_cartes:
40            cartes[c] += 1
41        return True
42    else:
43        return False
44
45 def nb_validee(j, v=False):
46     """
47     Compte le nombre total de cartes que le joueur j a déjà validé

```

```

46     Le paramètre v, s'il est a True, compte une carte de plus si le joueur a
47     ↳ déjà joué l'action validé
48     """
49     t = 0
50     for i in range(7):
51         t += nb_cartes_validees(j, i)
52     if est_jouee_action(j, action.VALIDER) and v:
53         t += 1
54     return t
55
56 def possede_abs(
57     g,
58     s=True,
59     defausse=[0, 0, 0, 0, 0, 0, 0],
60     add_m=[0, 0, 0, 0, 0, 0, 0],
61     add_a=[0, 0, 0, 0, 0, 0, 0],
62 ):
63     """
64     Paramètres d'entrées :
65     g -> la geicha à tester
66     s -> si True, prend en compte la carte secrète si on en a validé une
67     defausse -> la défausse à prendre en compte
68     add_m -> les cartes que l'on rajoute à soi-même pour faire une
69     ↳ simulation
70     add_a -> les cartes que l'on rajoute à l'adversaire pour faire une
71     ↳ simulation
72
73     Sortie : Renvoie qui va posséder la geicha à la fin de la manche :
74     1 -> si c'est moi qui la possède
75     -1 -> si c'est l'adversaire
76     0 -> si le résultat n'est pas encore défini
77
78     """
79     global MOI, ADV, sec
80     nb_cartes_jeu = valeur(g) - defausse[g]
81     cartes_v_m = nb_cartes_validees(MOI, g) + add_m[g] #Mes cartes validées
82     cartes_v_a = nb_cartes_validees(ADV, g) + add_a[g] #Les cartes validées
83     ↳ par l'adversaires visibles
84     nb_r_m = 8 - nb_validee(MOI, True) #Le nombre de cartes que je peux encore
85     ↳ valider
86     nb_r_a = 8 - nb_validee(ADV) #Le nombre de cartes que l'adversaire peut
87     ↳ encore valider (+ sa carte secrète)
88     for i in add_m:
89         nb_r_m -= i
90     for i in add_a:
91         nb_r_a -= i
92     diff = cartes_v_m - cartes_v_a
93     if s and sec == g:
94         cartes_v_m += 1
95     majorite = [0, 0] #Le nombre de cartes qu'il faut avoir pour obtenir la
96     ↳ majorité absolue
97     if possession_geisha(g) == joueur.EGALITE:

```

```

91     if nb_cartes_jeu in [0, 1]:
92         majorite = [1, 1]
93     elif nb_cartes_jeu in [2, 3]:
94         majorite = [2, 2]
95     else:
96         majorite = [3, 3]
97
98     tab_maj = [[0, 1], [1, 1], [1, 2], [2, 2], [2, 3], [3, 3]]
99     majorite = tab_maj[nb_cartes_jeu]
100     if possession_geisha(g) == ADV: #On inverse si c'est l'adversaire qui
101     ↳ a la possession
102         majorite[0], majorite[1] = majorite[1], majorite[0]
103     if cartes_v_m >= majorite[0]:
104         return 1
105     elif cartes_v_a >= majorite[1]:
106         return -1
107     elif nb_r_m + diff < 0 or (nb_r_m + diff == 0 and possession_geisha(g) ==
108     ↳ ADV): #Si je ne peux pas placer assez de cartes pour le dépasser
109         print("Special adv")
110         return -1
111     elif nb_r_a - diff < 0 or (nb_r_a - diff == 0 and possession_geisha(g) ==
112     ↳ MOI): #S'il ne peut pas placer assez de cartes pour me dépasser
113         print("Special moi")
114         return 1
115     else:
116         return 0
117
118 def possede_relatif(
119     g,
120     s=True,
121     defausse=[0, 0, 0, 0, 0, 0, 0],
122     add_m=[0, 0, 0, 0, 0, 0, 0],
123     add_a=[0, 0, 0, 0, 0, 0, 0],
124 ):
125     """
126     Paramètres d'entrées :
127     g -> la geicha à tester
128     s -> si True, prend en compte la carte secrète si on en a validé une
129     defausse -> la défausse à prendre en compte
130     add_m -> les cartes que l'on rajoute à soi-même pour faire une
131     ↳ simulation
132     add_a -> les cartes que l'on rajoute à l'adversaire pour faire une
133     ↳ simulation
134
135     Sortie : Renvoie qui possède la geicha actuellement :
136     1 -> si c'est moi qui la possède
137     -1 -> si c'est l'adversaire
138     0 -> si il y a égalité
139
140     """
141     global MOI, ADV, sec

```

```

138 if (possede_abs(g,s,defausse,add_m,add_a) != 0):#Si c'est vrai absolument,
139     ↪ on ne regarde même pas relativement
140     return possede_abs(g,s,defausse,add_m,add_a)
141 cartes_v_m = nb_cartes_validees(MOI, g) + add_m[g]
142 cartes_v_a = nb_cartes_validees(ADV, g) + add_a[g]
143 if s and sec == g:
144     cartes_v_m += 1
145 if cartes_v_m > cartes_v_a or (
146     cartes_v_m == cartes_v_a and possession_geisha(g) == MOI
147 ):
148     return 1
149 elif cartes_v_m < cartes_v_a or (
150     cartes_v_m == cartes_v_a and possession_geisha(g) == ADV
151 ):
152     return -1
153 else:
154     return 0
155 def simul_points(
156     s=True,
157     de=True,
158     add_m=[0, 0, 0, 0, 0, 0, 0],
159     add_a=[0, 0, 0, 0, 0, 0, 0],
160     defau=[0, 0, 0, 0, 0, 0, 0],
161     relatif=False,
162 ):
163     """
164     Paramètres d'entrées :
165     s -> si True, prend en compte la carte secrète si on en a validé une
166     de -> si True, prend en compte la defausse
167     add_m -> les cartes que l'on rajoute à soi-même pour faire une
168     ↪ simulation
169     add_a -> les cartes que l'on rajoute à l'adversaire pour faire une
170     ↪ simulation
171     defau -> la defausse que l'on rajoute pour faire une simulation
172     relatif -> si vrai, fait la simulation avec la fonction possede_relatif
173
174     Sortie : Un tableau contenant dans la première case le score que l'on a
175     et dans la seconde le score qu'a l'adversaire après simulation
176     """
177     score = [0, 0]
178     if de:
179         global defausse
180     else:
181         defausse = defau
182     for i in range(7):
183         if relatif:
184             a = possede_relatif(i, s, defausse, add_m, add_a)
185         else:
186             a = possede_abs(i, s, defausse, add_m, add_a)
187         if a == 1:
188             score[0] += valeur(i)

```

```

187         elif a == -1:
188             score[1] += valeur(i)
189         if score[0] >= 11 and not relatif: #Boost de points si on est sûr de
190             ↪ gagner
191             score[0] = 10000
192         elif score[1] >= 11 and not relatif: #Malus de points si on est sûr de
193             ↪ perdre
194             score[1] = 1000
195         return score

```

1.2 Fonction principale jouer_tour

Initialisation

```

195 MOI = 0
196 ADV = 0
197 sec = -1 #La carte secrete que l'on valide
198 defausse = [0 for _ in range(7)]
199 cartes = [0 for _ in range(7)]
200 nb_manche = -1
201
202 # Fonction appelee au debut du jeu
203 def init_jeu():
204     global MOI
205     global ADV
206     MOI = id_joueur()
207     ADV = id_adversaire()
208     print(MOI, "debut jeu")
209
210 # Fonction appelée au debut du tour
211 def jouer_tour():
212     print("C'est mon tour")
213     t1 = time.time()
214     global cartes, nb_manche, sec, defausse
215     l_cartes = cartes_en_main()
216     l_cartes.sort(reverse=True)#Tri des cartes en commençant par la plus
217     ↪ forte
218     action_non_faite = True
219
220     if not(nouvelle_manche()):
221         p = carte_piochee()
222         cartes[p] += 1

```

1.2.1 Astuces

Choix trois

```

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

```

```

229         if (cartes[i] >= 3):
230             cartes[i] -= 3
231             e = action_choix_trois(i, i, i)
232             print("Triple choix !!")
233             action_non_faute = False
234             break

```

Choix paquets

```

236     #Si on a deux paquets identiques
237     if not est_jouee_action(MOI, action.CHOIX_PAQUETS) and action_non_faute:
238         t = []
239         for i in range(len(cartes)):
240             if cartes[i] >= 2 and possession_geisha(i) != ADV:
241                 t.append(i)
242         while len(t) >= 2 and action_non_faute:
243             add = [[0 for _ in range(7)] for _ in range(2)]
244             add[0][t[0]] += 1
245             add[0][t[1]] += 1
246             add[1][t[0]] += 1
247             add[1][t[1]] += 1
248             if possede_abs(t[0], add_m=add[0], add_a=add[1]) == -1: #Vérifie
249                 ↳ que ça ne fait pas gagner des points à l'adversaire
250                 t.pop(0)
251             elif possede_abs(t[1], add_m=add[0], add_a=add[1]) == -1:
252                 t.pop(1)
253             else:
254                 cartes[t[0]] -= 2
255                 cartes[t[1]] -= 2
256                 e = action_choix_paquets(t[0], t[1], t[0], t[1])
257                 print("Deux paquets identiques !")
258                 action_non_faute = False

```

Defausser

```

259     #Défausse idéale
260     if not est_jouee_action(MOI, action.DEFAUSSER) and action_non_faute:
261         for i in range(len(l_cartes)): #Pour toutes les permutations de cartes
262             ↳ possibles
263             for j in range(i + 1, len(l_cartes)):
264                 if action_non_faute:
265                     add = [0, 0, 0, 0, 0, 0, 0]
266                     add[l_cartes[i]] += 1
267                     add[l_cartes[j]] += 1
268                     if (
269                         possede_abs(l_cartes[i], defausse=add) == 1
270                         and possede_abs(l_cartes[j], defausse=add) == 1
271                         and possede_abs(l_cartes[i]) != 1
272                         and possede_abs(l_cartes[j]) != 1
273                         and l_cartes[i] != l_cartes[j]
274                     ): #Si on prend la possession des 2 geichas
275                         cartes[l_cartes[i]] -= 1
276                         cartes[l_cartes[j]] -= 1

```

```

276         e = action_defausser(l_cartes[i], l_cartes[j])
277         print("Defausse tres rentable")
278         action_non_faute = False
279         defausse = add
280         break
281     if not action_non_faute:
282         break

```

Valider

```

284     #Action valider
285     if not est_jouee_action(MOI, action.VALIDER) and action_non_faute:
286         non = set()
287         continuer = True
288         while len(non) < 7 and action_non_faute and continuer:
289             lv = [-1]
290             for i in range(len(cartes) - 1, -1, -1):
291                 #On regarde toutes les cartes de mêmes valeurs que l'on
292                 ↳ possède, que l'on a pas éliminé
293                 # et dont la possession est relative
294                 if (
295                     cartes[i] != 0
296                     and possede_abs(i) == 0
297                     and valeur(i) > valeur(lv[0])
298                     and not (i in non)
299                 ):
300                     lv = [i]
301                 elif (
302                     cartes[i] != 0
303                     and possede_abs(i) == 0
304                     and valeur(i) == valeur(lv[0])
305                     and not (i in non)
306                 ):
307                     lv.append(i)
308             if len(lv) == 1 and lv[0] != -1:
309                 sec = lv[0]
310                 if possession_geisha(lv[0]) != ADV or possede_abs(lv[0]) == 1:
311                     #Si elle n'appartient pas à l'adversaire ou que on aura
312                     ↳ l'avantage après
313                     cartes[lv[0]] -= 1
314                     e = action_valider(lv[0])
315                     print("Je valide !", lv[0])
316                     action_non_faute = False
317                 else:
318                     non.add(lv[0])
319                     sec = -1
320             elif len(lv) > 1: #S'il y en a plusieurs
321                 for c in lv:
322                     if (
323                         possession_geisha(c) == MOI
324                         ): #On valide en priorité les cartes dont on a l'avantage
325                         cartes[c] -= 1

```

```

325         sec = c
326         e = action_valider(c)
327         print("Je valide !", c)
328         action_non_faites = False
329         break
330     if action_non_faites:
331         for c in lv:
332             if possession_geisha(c) == joueur.EGALITE:
333                 #Puis les cartes dont personne n'a d'avantage
334                 cartes[c] -= 1
335                 sec = c
336                 e = action_valider(c)
337                 print("Je valide !", c)
338                 action_non_faites = False
339                 break
340     if action_non_faites:
341         for c in lv:
342             sec = c
343             if possede_abs(c) == 1:
344                 #Puis si c'est l'adversaire qui a l'avantage et
345                 ↳ que on est pas sûr de gagner la carte après
346                 ↳ validation
347                 # On préféreras refaire un autre tour
348                 cartes[c] -= 1
349                 e = action_valider(lv[0])
350                 print("Je valide !", lv[0])
351                 action_non_faites = False
352                 break
353             else:
354                 sec = -1
355                 non.add(c)
356         continuer = False
357     if action_non_faites:
358         print("Et c'est parti pour un autre tour")
359     if action_non_faites: #Si toutes les cartes ne semblent pas
360     ↳ 'rentables', on valide la plus forte
361     for i in range(len(cartes) - 1, -1, -1):
362         if cartes[i] != 0:
363             cartes[i] -= 1
364             sec = i
365             e = action_valider(i)
366             action_non_faites = False
367             break

```

1.2.2 Simulation dans le cas échéant

Choix trois

```

367     # Fait le choix triple en fonction d'un algo min-max partiel
368     if not est_jouee_action(MOI, action.CHOIX_TROIS) and action_non_faites:
369         choix_f = []
370         maxi = -inf

```

```

371     for i in range(len(l_cartes)):
372         for j in range(i + 1, len(l_cartes)):
373             for l in range(j + 1, len(l_cartes)): #Pour toutes les
374             ↳ permutations possibles
375             mini = inf
376             for m in range(3): #Pour chaque'un des placements des
377             ↳ cartes
378             #Ajouts des cartes pour simulation
379             add_m = [0 for _ in range(7)]
380             add_a = [0 for _ in range(7)]
381             if m == 0:
382                 add_m[l_cartes[i]] += 1
383                 add_m[l_cartes[j]] += 1
384                 add_a[l_cartes[l]] += 1
385             elif m == 1:
386                 add_m[l_cartes[i]] += 1
387                 add_a[l_cartes[j]] += 1
388                 add_m[l_cartes[l]] += 1
389             else:
390                 add_a[l_cartes[i]] += 1
391                 add_m[l_cartes[j]] += 1
392                 add_m[l_cartes[l]] += 1
393
394             res = simul_points(add_m=add_m, add_a=add_a,
395             ↳ relatif=True)
396             if res[0] - res[1] < mini: #On regarde la pire
397             ↳ différence
398             mini = res[0] - res[1]
399             if mini > maxi: #On regarde le meilleur choix parmi
400             ↳ toutes les simulations
401             choix_f = [l_cartes[i], l_cartes[j], l_cartes[l]]
402             score = res
403             maxi = mini
404
405     print("Choix trois par simulation :", score)
406     for c in choix_f:
407         cartes[c] -= 1
408     e = action_choix_trois(choix_f[0], choix_f[1], choix_f[2])
409     action_non_faites = False

```

Défausser

```

407     #Action defausser 2
408     if not est_jouee_action(MOI, action.DEFAUSSER) and action_non_faites:
409         interessante = -1
410         for i in range(len(l_cartes)):
411             for j in range(i + 1, len(l_cartes)): #Pour toutes les
412             ↳ permutations possibles
413             if action_non_faites:
414                 add = [0, 0, 0, 0, 0, 0, 0]
415                 add[l_cartes[i]] += 1
416                 add[l_cartes[j]] += 1
417                 if (

```

```

417         possede_abs(l_cartes[i], defausse=add) == 1
418         and possede_abs(l_cartes[j], defausse=add) == 1
419     ):#Defausse deux cartes identiques ou ininteressantes
420         cartes[l_cartes[i]] -= 1
421         cartes[l_cartes[j]] -= 1
422         e = action_defausser(l_cartes[i], l_cartes[j])
423         action_non_faute = False
424         defausse = add
425         break
426     elif (
427         possede_abs(l_cartes[j], defausse=add) == 1
428         and possede_abs(l_cartes[j]) != -1
429         and l_cartes[j] > interessante
430     ):#On stocke la carte qui nous semble interessante
431         interessante = l_cartes[j]
432     elif (
433         possede_abs(l_cartes[i], defausse=add) == 1
434         and possede_abs(l_cartes[i]) != -1
435         and l_cartes[i] > interessante
436     ):
437         interessante = l_cartes[i]
438     if not action_non_faute:
439         break
440 if action_non_faute and interessante != -1:#Si on en a trouvé une
↳ interessante, on la valide avec une autre aléatoire
441     l_cartes.remove(interessante)
442     cartes[interessante] -= 1
443     cartes[l_cartes[-1]] -= 1
444     defausse[interessante] += 1
445     defausse[l_cartes[-1]] += 1
446     e = action_defausser(interessante, l_cartes[-1])
447     print("Defausse a demi interessante")
448     action_non_faute = False
449
450 #Defausse 3
451 if not est_jouee_action(MOI, action.DEFAUSSER) and action_non_faute:
452     choix = []
453     diff = -inf
454     for i in range(len(l_cartes)):
455         for j in range(i + 1, len(l_cartes)):#Toutes les permutations
456             add = [0, 0, 0, 0, 0, 0, 0]
457             add[l_cartes[i]] += 1
458             add[l_cartes[j]] += 1
459             res = simul_points(de=False, defau=add)
460             if res[0] - res[1] > diff:#On regarde le choix qui nous fait
↳ perdre le moins de points
461                 choix = [l_cartes[i], l_cartes[j]]
462                 diff = res[0] - res[1]
463
464     for i in range(2):
465         cartes[choix[i]] -= 1
466         defausse[choix[i]] += 1

```

```

467     e = action_defausser(choix[0], choix[1])
468     action_non_faute = False
469     print("Je defausse par simulation")

```

Choix paquets

```

473 #Choix paquets
474 if not est_jouee_action(MOI, action.CHOIX_PAQUETS) and action_non_faute:
475     maximum = max(cartes)
476     if maximum == 3 or maximum == 4:#On n'a pas le choix
477         for i in range(4):
478             cartes[l_cartes[i]] -= 1
479             e = action_choix_paquets(l_cartes[0], l_cartes[1], l_cartes[2],
↳ l_cartes[3])
480             action_non_faute = False
481             print("Dernier choix de paquets force (sans reel choix)")
482     elif maximum == 2:# 2 cartes identiques
483         num = -1
484         for l in range(len(cartes)):
485             if cartes[l] == 2:
486                 num = l
487
488         liste_d = l_cartes.copy()
489         add = [[[0 for _ in range(7)] for _ in range(2)] for _ in
↳ range(2)]#Liste de toutes les possibilités
490         liste_d.remove(num)
491         liste_d.remove(num)
492         #Remplissage de la liste
493         add[l][0][num] += 2
494         add[l][1][liste_d[0]] += 1
495         add[l][1][liste_d[1]] += 1
496         add[0][0][num] += 1
497         add[0][1][num] += 1
498         add[0][0][liste_d[0]] += 1
499         add[0][1][liste_d[1]] += 1
500
501
502     if possede_abs(
503         num, defausse=defausse, add_m=add[0][0], add_a=add[0][1]
504     ) == -1 and possede_abs(
505         num, defausse=defausse, add_m=add[1][0], add_a=add[1][1]
506     ):
507         e = action_choix_paquets(num, num, liste_d[0], liste_d[1])
508         action_non_faute = False
509         print("Choix paquets optimal avec les deux identiques du meme
↳ cote ")
510     elif possede_abs(num, defausse=defausse, add_m=add[0][0],
↳ add_a=add[0][1]) == 1:
511         e = action_choix_paquets(num, liste_d[0], num, liste_d[1])
512         action_non_faute = False
513         print("Choix paquets optimal avec les deux identiques dans des
↳ paquets differents ")
514     else:

```

```

515 liste_d = l_cartes.copy()
516 add = [[[0 for _ in range(7)] for _ in range(2)] for _ in
      ↪ range(3)]
517 #Remplissage de la liste
518 add[0][0][liste_d[0]] += 1
519 add[0][1][liste_d[1]] += 1
520 add[0][0][liste_d[2]] += 1
521 add[0][1][liste_d[3]] += 1
522 add[1][0][liste_d[0]] += 1
523 add[1][1][liste_d[2]] += 1
524 add[1][0][liste_d[1]] += 1
525 add[1][1][liste_d[3]] += 1
526 add[2][0][liste_d[0]] += 1
527 add[2][1][liste_d[2]] += 1
528 add[2][0][liste_d[1]] += 1
529 add[2][1][liste_d[3]] += 1
530 #Simulation min-max partielle
531 if action_non_faite:
532     choix_f = []
533     maxi = -inf
534     for i in range(len(add)):
535         mini = inf
536         for j in range(2):
537             l = 1 #1 si j = 0; 0 si j = 1
538             if j == 1:
539                 l = 0
540             res = simul_points(add_m=add[i][j], add_a=add[i][1])
541             if res[0] - res[1] < mini:
542                 mini = res[0] - res[1]
543         if mini > maxi:
544             #Ajouts des cartes
545             choix_f = []
546             for p in range(2):
547                 for c in range(7):
548                     for _ in range(add[i][p][c]):
549                         choix_f.append(c)
550             assert len(choix_f) == 4, "Mauvais nombre de cartes"
551             maxi = mini
552     e = action_choix_paquets(choix_f[0], choix_f[1], choix_f[2],
      ↪ choix_f[3])
553     action_non_faite = False
554     print("Choix paquets apres simulation")
555
556 if action_non_faite:
557     print("J'ai une erreur, aucune action n'a ete faite")
558 elif e != error.OK:
559     print("J'ai essayer de faire une action mais j'ai eu l'erreur", e)
560 else:
561     print("J'ai bien fait mon action en", time.time() - t1)
562     print(cartes)
563     print()

```

1.3 Répondre à l'action 3

```

565 # Fonction appelee lors du choix entre les trois cartes lors de l'action de
566 # l'adversaire (cf tour_precedent)
567 def repondre_action_choix_trois():
568     nouvelle_manche()
569     print("Repondre action 3")
570     choix = []
571     maxi = []
572     tour_p = tour_precedent()
573     lc = [tour_p.c1, tour_p.c2, tour_p.c3] #Liste des cartes possibles
574     for i in range(3):
575         maxi.append(lc[i])
576         choix.append(i)
577     if maxi[0] == maxi[1] and maxi[1] == maxi[2]:
578         print("Trois cartes identiques")
579         e = repondre_choix_trois(choix[0])
580     else:
581         res = []
582         choix_m = 0
583         diff = -inf
584         for l in range(3): #Pour chaque cartes possibles
585             liste_cartes = lc.copy()
586             liste_cartes.remove(maxi[l])
587             add_m = [0, 0, 0, 0, 0, 0, 0]
588             add_a = [0, 0, 0, 0, 0, 0, 0]
589             add_m[maxi[l]] += 1
590             add_a[liste_cartes[0]] += 1
591             add_a[liste_cartes[1]] += 1
592             res.append(simul_points(add_m=add_m, add_a=add_a,
      ↪ relatif=True)) #Simulations
593             if res[l][0] - res[l][1] > diff:
594                 diff = res[l][0] - res[l][1]
595                 choix_m = choix[l]
596         e = repondre_choix_trois(choix_m)
597     print(len(choix), e)
598     print()

```

1.4 Répondre à l'action 4

```

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

```



```

610 else:
611     res = []
612     choix_m = -1
613     diff = -inf
614     for l in range(0, 3, 2): #l = 0 ou 2
615         liste_cartes = list(lc)
616         add_m = [0, 0, 0, 0, 0, 0, 0]
617         add_a = [0, 0, 0, 0, 0, 0, 0]
618         add_m[liste_cartes.pop(1)] += 1 #Les deux cartes donnés à
        ↪ l'adversaires (0,1 ou 2,3)
619         add_m[liste_cartes.pop(1)] += 1
620         add_a[liste_cartes[0]] += 1
621         add_a[liste_cartes[1]] += 1
622         res.append(simul_points(add_m=add_m, add_a=add_a, relatif=True))
623         i = 0
624         if l == 2:
625             i = 1
626         if res[i][0] - res[i][1] > diff:
627             diff = res[i][0] - res[i][1]
628             choix_m = i
629
630     print(choix_m)
631     e = repondre_choix_paquets(choix_m)
632     print("Resultat simulation :", res)
633     print("Erreur :", e)
634     print()
635
636 # Fonction appelee a la fin du jeu
637 def fin_jeu():
638     print("Fin jeu")

```

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

```

1 import itertools as it
2 from tqdm import tqdm
3 import os
4 from copy import deepcopy
5 import time
6
7 # paquets = [5, 5, 5, 3, 3, 3, 6, 6, 6, 6, 6, 0, 5, 0, 1, 2, 1, 2, 4, 4, 4]
8 # paquets = [2, 4, 5, 3, 2, 1, 0, 3, 6, 4, 6, 5, 1, 6, 6, 5, 4, 6, 5, 3, 0]
9 paquets = [0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 6, 6, 6, 6, 3, 4, 5, 5, 6]
10
11
12 def possiblite(p, n):
13     a = it.combinations(p, n)
14     s = set()
15     for i in a:
16         s.add(i)
17     return s
18
19
20 def C():
21     if os.path.exists("save_graphe.txt"):
22         print("Récupération du fichier en cours")
23         fichier = open("save_graphe.txt", "r")
24         temps = int(fichier.readline().strip())
25         cpt = int(fichier.readline().strip())
26         cpt_fin = int(fichier.readline().strip())
27         tour_boucle = int(fichier.readline().strip())
28         nb_pile = int(fichier.readline().strip())
29         pile = []
30         for _ in range(nb_pile):
31             etat = []
32             for _ in range(2):
33                 p1 = fichier.readline().strip().split(" ")
34                 if p1 == [""]:
35                     p1 = []
36                 for elem in range(len(p1)):
37                     p1[elem] = int(p1[elem])
38                 etat.append(p1)
39             for _ in range(2):
40                 p1 = fichier.readline().strip().split(" ")
41                 s1 = set()
42                 if p1 == [""]:
43                     p1 = []
44                 for elem in range(len(p1)):
45                     try:
46                         s1.add(int(p1[elem]))
47                     except:
48                         print(p1)
49                         exit()
50                 etat.append(s1)

```



```

51         etat.append(int(fichier.readline().strip()))
52         etat.append(int(fichier.readline().strip()))
53         pile.append(etat)
54     pbar = tqdm(initial=tour_boucle, total=229249440000)
55     print("Récupération du fichier terminé")
56 else:
57     pbar = tqdm(total=229249440000)
58     temps = 0
59     tour_boucle = 0
60     cpt = 0
61     cpt_fin = 0
62     pile = [
63         [
64             paquets[:6],
65             paquets[6:12],
66             set([1, 2, 3, 4]),
67             set([1, 2, 3, 4]),
68             12,
69             1,
70             set(),
71         ]
72     ]
73     temps1 = int(time.time())
74     while len(pile) != 0:
75         """
76         if tour_boucle % 10000000 == 0:
77             print(tour_boucle, len(pile))
78         """
79         etat = pile.pop()
80         p1 = etat[0] # paquet 1
81         p2 = etat[1] # paquet 2
82         t1 = etat[2]
83         t2 = etat[3]
84         tour = etat[4]
85         cst = etat[5]
86         if tour == 20:
87             cpt_fin += 1
88         elif tour % 2 == 0:
89             p1.append(paquets[tour])
90             assert len(p1) <= 7, (p1, tour)
91             for i in t1:
92                 a = possiblite(p1, i)
93                 for p in a:
94                     p1suiv = deepcopy(p1)
95                     t1suiv = deepcopy(t1)
96                     t1suiv.remove(i)
97                     for carte in p:
98                         p1suiv.remove(carte)
99                     cst1 = cst
100                     if i == 3:
101                         cst1 *= 3
102                     if i == 4:

```

```

103                         cst1 *= 6
104                     assert len(p1suiv) <= 7, (p1suiv, p1, tour_boucle, p)
105                     pile.append(
106                         [p1suiv, deepcopy(p2), t1suiv, deepcopy(t2), tour + 1,
107                          ↳ cst1]
108                     )
109                     cpt += cst1
110 else:
111     p2.append(paquets[tour])
112     assert len(p2) <= 7, p2
113     for i in t2:
114         a = possiblite(p2, i)
115         for p in a:
116             p2suiv = deepcopy(p2)
117             t2suiv = deepcopy(t2)
118             t2suiv.remove(i)
119             for carte in p:
120                 p2suiv.remove(carte)
121             cst2 = cst
122             if i == 3:
123                 cst2 *= 3
124             if i == 4:
125                 cst2 *= 6
126             assert len(p1) <= 7, p1
127             pile.append(
128                 [deepcopy(p1), p2suiv, deepcopy(t1), t2suiv, tour + 1,
129                  ↳ cst2]
130             )
131             cpt += cst2
132     tour_boucle += 1
133     pbar.update(1)
134     if tour_boucle % 10000000 == 0:
135         print("Sauvegarde en cours...")
136         # print(pile)
137         fichier = open("save_graphe.txt", "w")
138         fichier.write(str(temps + int(time.time()) - temps1) + "\n")
139         fichier.write(str(cpt) + "\n")
140         fichier.write(str(cpt_fin) + "\n")
141         fichier.write(str(tour_boucle) + "\n")
142         fichier.write(str(len(pile)) + "\n")
143         for etat_p in pile:
144             for l in range(4):
145                 for p1_c in etat_p[l]:
146                     fichier.write(str(p1_c) + " ")
147                     fichier.write("\n")
148                     fichier.write(f"{etat_p[4]}\n{etat_p[5]}\n")
149             fichier.close()
150             print("Sauvegarde terminée")
151     pbar.close()
152     print("Enregistrement")
153     fichier = open("fin.txt", "w")
154     fichier.write(

```

```

153     "Temps en secondes : " + str(temps + int(time.time()) - temps1) + "\n"
154 )
155 fichier.write("Nombre de noeuds dans le graphe maximal : " + str(cpt) +
    ↪ "\n")
156 fichier.write("Nombre d'états finaux : " + str(cpt_fin) + "\n")
157 fichier.write("Nombre de tours de boucle : " + str(tour_boucle) + "\n")
158 fichier.close()
159 print("Terminé !")
160
161 return cpt
162
163
164 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

```

1  #include "api.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <stdbool.h>
5  #include "calcul_score.h"
6  #include <sys/time.h>
7  #include "convertir.h"
8
9  typedef struct etat
10 {
11     int *valide_adv; // Les cartes déjà validées par mon adversaire
12     int *valide_moi; // Les caryes déjà validées par moi
13     int *avantage; // -1 si l'avantage est à mon adversaire +1 si c'est moi
    ↪ 0 sinon (en fonction de chaque couleur)
14 } ETAT;
15
16 typedef struct game
17 {
18     int *cartes;
19     int *restantes;
20     int en_main;
21     int nb_restantes;
22     int valide;
23     int defausse1;
24     int defausee2;
25     bool *act_poss;
26     ETAT *etat;
27 } GAME; // L'état actuel du jeu
28
29 typedef struct marqueurs
30 {
31     int k; // la taille de pointeurs // le nombre de cartes choisies
32     int n; // le nombre de cartes dans lequel on choisit
33     int *cartes; // Les cartes dans lequel on choisit
34     int *pointeurs; // Les cartes choisies
35 } marq;
36
37 typedef struct coup
38 {
39     int action;
40     int *cartes;
41 } COUP;
42
43 GAME g;
44 int manche_accu = -1;
45 joueur moi;

```

```

46 joueur adv;
47 SIX*** donnees;
48
49 // LES CONSTANTES
50 int valeur_couleur[7] = {2, 2, 2, 3, 3, 4, 5}; // La
51   ↳ valeur des couleurs = au nombre de cartes
52 int permu_trois[3][2] = {{1, 2}, {0, 2}, {0, 1}}; // Les deux
53   ↳ cartes non choisis
54 int permu_paquet[2][4] = {{0, 1, 2, 3}, {2, 3, 0, 1}}; // Les
55   ↳ permutations pour deux paquets
56 int nb_cartes_par_action[4][2] = {{1, 0}, {0, 0}, {2, 1}, {2, 2}}; // Le
57   ↳ nombre de cartes validés par action en fonction des joueurs
58
59 long t1;
60 long currenttime()
61 {
62     /*Renvoie le temps actuel en millisecondes*/
63     struct timeval tp;
64     gettimeofday(&tp, NULL);
65     return tp.tv_sec * 1000 + tp.tv_usec / 1000;
66 }
67
68 void debug_cartes(int nb, int *ens_cartes, char *nom)
69 {
70     /*Affiche le paquet de carte ens_cartes comprenant nb cases en affichant
71     ↳ nom avant*/
72     printf("%s : ", nom);
73     for (int i = 0; i < nb; i++)
74     {
75         printf("%d ", ens_cartes[i]);
76     }
77     printf("\n");
78     fflush(stdout);
79 }
80
81 void toutes_les_cartes(int *ens_cartes)
82 {
83     /*Initialise le tableau ens_cartes aux valeurs contenues dans
84     ↳ valeur_couleur (variable globale)*/
85     for (int i = 0; i < 7; i++)
86     {
87         ens_cartes[i] = valeur_couleur[i];
88     }
89 }
90
91 void aucune_carte(int *ens_cartes)
92 {
93     /*Initialise le tableau ens_cartes à 0*/
94     for (int i = 0; i < 7; i++)
95     {
96         ens_cartes[i] = 0;
97     }
98 }

```

```

92 }
93
94 void update_cartes_valides(void)
95 {
96     /*Met à jour les cartes validées et restantes*/
97     toutes_les_cartes(g.restantes);
98     g.nb_restantes = 0;
99     for (int i = 0; i < 7; i++)
100     {
101         g.etat->valide_moi[i] = nb_cartes_validees(moi, i);
102         g.etat->valide_adv[i] = nb_cartes_validees(adv, i);
103         g.restantes[i] = g.restantes[i] - g.etat->valide_moi[i] -
104             ↳ g.etat->valide_adv[i] - g.cartes[i];
105         g.nb_restantes += g.restantes[i];
106     }
107     if (!(g.act_poss[0])) // Carte validée secretement
108     {
109         g.restantes[g.valide] -= 1;
110         g.etat->valide_moi[g.valide] += 1;
111     }
112     if (!(g.act_poss[1])) // Cartes defaussees
113     {
114         g.restantes[g.defausse1] -= 1;
115         g.restantes[g.defausee2] -= 1;
116         g.nb_restantes -= 2;
117     }
118 }
119
120 void update(bool new_c)
121 {
122     /*Met à jour l'état du jeu au début d'un tour*/
123     joueur poss;
124     if (manche_accu != manche()) // Pour une nouvelle manche
125     {
126         g.valide = -1;
127         g.en_main = 0;
128         g.defausse1 = -1;
129         g.defausee2 = -1;
130         for (int i = 0; i < 7; i++) // On regarde les avantages de chaque
131             ↳ couleur
132         {
133             poss = possession_geisha(i);
134             if (poss == moi)
135             {
136                 g.etat->avantage[i] = 1;
137             }
138             else if (poss == EGALITE)
139             {
140                 g.etat->avantage[i] = 0;
141             }
142             else
143             {
144

```

```

142         g.etat->avantage[i] = -1;
143     }
144 }
145 for (int i = 0; i < 4; i++) // Toutes les actions sont à nouveau
    ↳ disponible
146 {
147     g.act_poss[i] = true;
148 }
149 int_array lc = cartes_en_main();
150 aucune_carte(g.cartes);
151 for (int j = 0; j < lc.length; j++) // Mets à jour les cartes en main
152 {
153     g.cartes[lc.items[j]] += 1;
154     g.en_main += 1;
155 }
156 update_cartes_valides();
157 manche_accu += 1;
158 }
159 else
160 {
161     if (new_c) // Si c'est juste un nouveau tour, ajoute la carte piochée
        ↳ (si elle existe (en fonction de new_c))
162     {
163         int pioche = carte_piochee();
164         g.cartes[pioche] += 1;
165         g.en_main += 1;
166     }
167     update_cartes_valides();
168 }
169 }
170
171 void joue_valide(int c)
172 {
173     /*Joue l'action valider avec la carte c*/
174     g.act_poss[0] = false;
175     g.cartes[c] -= 1;
176     g.en_main -= 1;
177     g.valide = c;
178     error e = action_valider(c);
179     if (e == OK)
180     {
181         printf("Action valide carte : %d\n", c);
182     }
183     else
184     {
185         printf("!!!!!!!!!!!!!!!!!!!! ERREUR !!!!!!!!!!!!!: %d\n", e);
186         printf("Action valide carte : %d\n", c);
187     }
188 }
189
190 void joue_defausse(int d1, int d2)
191 {

```

```

192     /*Joue l'action défausser avec les cartes d1 et d2*/
193     g.act_poss[1] = false;
194     g.cartes[d1] -= 1;
195     g.cartes[d2] -= 1;
196     g.en_main -= 2;
197     g.defausse1 = d1;
198     g.defausse2 = d2;
199     error e = action_defausser(d1, d2);
200     if (e == OK)
201     {
202         printf("Action defausser cartes : %d, %d\n", d1, d2);
203     }
204     else
205     {
206         printf("!!!!!!!!!!!!!!!!!!!! ERREUR !!!!!!!!!!!!!: %d\n", e);
207         printf("Action defausser cartes : %d, %d\n", d1, d2);
208     }
209 }
210
211 void joue_trois(int c1, int c2, int c3)
212 {
213     /*Joue l'action 3 avec les cartes c1, c2 et c3*/
214     g.act_poss[2] = false;
215     g.cartes[c1] -= 1;
216     g.cartes[c2] -= 1;
217     g.cartes[c3] -= 1;
218     g.en_main -= 3;
219     error e = action_choix_trois(c1, c2, c3);
220     if (e == OK)
221     {
222         printf("Action triple choix cartes : %d, %d, %d\n", c1, c2, c3);
223     }
224     else
225     {
226         printf("!!!!!!!!!!!!!!!!!!!! ERREUR !!!!!!!!!!!!!: %d\n", e);
227         printf("Action triple choix cartes : %d, %d, %d\n", c1, c2, c3);
228     }
229 }
230
231 void joue_quatre(int c11, int c12, int c21, int c22)
232 {
233     /*Joue l'action des paquets avec les cartes c11 et c12 d'un côté, et les
        ↳ cartes c21 et c22 de l'autre*/
234     g.act_poss[3] = false;
235     g.cartes[c11] -= 1;
236     g.cartes[c12] -= 1;
237     g.cartes[c21] -= 1;
238     g.cartes[c22] -= 1;
239     g.en_main -= 4;
240     error e = action_choix_paquets(c11, c12, c21, c22);
241     if (e == OK)
242     {

```

```

243     printf("Action choix paquets cartes : %d %d %d %d\n", c11, c12, c21,
244           ↪ c22);
245 }
246 else
247 {
248     printf("!!!!!!!!!!!!!!!!!!!! ERREUR !!!!!!!!!!!!!: %d\n", e);
249     printf("Action choix paquets cartes : %d, %d, %d, %d\n", c11, c12,
250           ↪ c21, c22);
251 }
252 }

```

3.1.2 Simulations

```

252 marq *init_marqueur(int k, int n, int *cartes)
253 {
254     /*Initialise le marqueur pour choisir k cartes parmi les n cartes dans
255     ↪ cartes*/
256     marq *m = malloc(sizeof(marq));
257     m->cartes = cartes;
258     m->k = k;
259     m->n = n;
260     if (k <= n)
261     {
262         m->pointeurs = malloc(k * sizeof(int));
263         int point = 0;
264         int carte = 0;
265         while (point < k && carte < 7)
266         {
267             // prend la première carte possible
268             for (int i = 0; i < cartes[carte] && point < k; i++)
269             {
270                 m->pointeurs[point++] = carte;
271             }
272             carte++;
273         }
274     }
275     else
276     {
277         debug_cartes(7, cartes, "Cartes initialisés : ");
278         m->pointeurs = NULL;
279     }
280     return m;
281 }
282 void choix_cartes(marq *m)
283 {
284     /*Le marqueur m propose un nouveau choix de cartes dans la case pointeur,
285     ↪ et NULL s'il en existe plus*/
286     int dernier_non_vider = m->k - 1;
287     int non_plein = 6;
288     int cpt;
289     int continuer = true;

```

```

289 // cherche la première case vide pour faire avancer un pointeur
290 while (dernier_non_vider >= 0 && continuer)
291 {
292     cpt = m->cartes[non_plein];
293     while (cpt > 0 && dernier_non_vider >= 0 && continuer)
294     {
295         if (m->pointeurs[dernier_non_vider] == non_plein)
296         {
297             cpt--;
298             dernier_non_vider--;
299         }
300         else
301         {
302             continuer = false;
303         }
304     }
305     non_plein--;
306 }
307
308 if (dernier_non_vider < 0) // il n'y a plus de cases vides
309 {
310     free(m->pointeurs);
311     m->pointeurs = NULL;
312 }
313 else
314 {
315     non_plein = m->pointeurs[dernier_non_vider] + 1;
316     while (m->cartes[non_plein] == 0)
317     {
318         non_plein++;
319     }
320     m->pointeurs[dernier_non_vider] = non_plein;
321     cpt = m->cartes[m->pointeurs[dernier_non_vider]] - 1;
322     dernier_non_vider++;
323     for (int i = 0; i < cpt && dernier_non_vider < m->k; i++)
324     { // On repositionne les pointeurs suivants
325         m->pointeurs[dernier_non_vider++] = non_plein;
326     }
327     non_plein++;
328     while (dernier_non_vider < m->k && non_plein < 7)
329     { // On décale tous les autres pointeurs
330         for (int i = 0; i < m->cartes[non_plein] && dernier_non_vider <
331           ↪ m->k; i++)
332         {
333             m->pointeurs[dernier_non_vider++] = non_plein;
334         }
335         non_plein++;
336     }
337     if (dernier_non_vider < m->k && non_plein >= 7) // Si les derniers
338     ↪ pointeurs n'ont plus la place
339     {
340         free(m->pointeurs);

```

```

339         m->pointeurs = NULL;
340     }
341 }
342
343
344 void free_marq(marq *m)
345 {
346     /*Déalloue la mémoire du marqueur m*/
347     if (m->pointeurs != NULL)
348     {
349         free(m->pointeurs);
350     }
351     free(m);
352 }
353
354 bool verification(int nb_cartes, int *cartes, int nb_restantes, int
↪ *restantes, int nb_selec, int *select, ETAT *etat, bool *act_oss_simu)
355 {
356     /*Vérifie si la distribution des cartes est possible
357     Entree : cartes : les cartes en main
358             restantes : les cartes que l'on a toujours pas vu
359             select : les cartes que l'on voudrait valider de notre coté
360             etat : l'état du jeu
361             act_oss_simu : les actions encore possible (après
↪             simulation)*/
362     int nb_moi_max = 0;
363     int nb_moi_min = 0;
364     for (int i = 0; i < 4; i++)
365     {
366         if (act_oss_simu[i])
367         {
368             nb_moi_max += nb_cartes_par_action[i][0];
369         }
370     }
371     int tour_a = tour();
372     int t; // Le nombre de cartes que l'on a pas encore vu et qui vont
↪ arriver dans notre main
373     if (moi == 1 && tour_a == 0)
374     {
375         t = 4;
376     }
377     else if (tour_a == 0 || tour_a == 1 || (tour_a == 2 && moi == 1))
378     {
379         t = 3;
380     }
381     else if (tour_a == 2 || tour_a == 3 || (tour_a == 4 && moi == 1))
382     {
383         t = 2;
384     }
385     else if (tour_a == 4 || tour_a == 5 || (tour_a == 6 && moi == 1))
386     {
387         t = 1;

```

```

388     }
389     else
390     {
391         t = 0;
392     }
393     if (nb_moi_max - t > 0)
394     {
395         nb_moi_min = nb_moi_max - t;
396     }
397     int nb_moi_borne_inf = 0;
398     int nb_moi_borne_sup = 0;
399     int restantes_d[7];
400     int cartes_d[7];
401     for (int c = 0; c < 7; c++)
402     {
403         restantes_d[c] = restantes[c];
404         cartes_d[c] = cartes[c];
405     }
406     for (int c = 0; c < nb_selec; c++)
407     {
408         restantes_d[select[c]] -= 1;
409         cartes_d[select[c]] -= 1;
410     }
411     for (int c = 0; c < 7; c++)
412     {
413         nb_moi_borne_sup += cartes[c] - cartes_d[c]; // On prend en priorité
↪ les cartes qui sont dans notre main
414         if (restantes_d[c] < 0)
415         {
416             nb_moi_borne_inf -= restantes_d[c]; // On prend en priorité les
↪ cartes qui ne sont pas dans notre main
417         }
418     }
419     if (nb_moi_borne_inf > nb_moi_max || nb_moi_borne_sup < nb_moi_min)
420     {
421         // printf("Coup impossible %d %d %d %d\n", nb_moi_borne_inf,
↪ nb_moi_max, nb_moi_borne_sup, nb_moi_min);
422         return false;
423     }
424     else
425     {
426         return true;
427     }
428 }
429
430 float simulation_coup(int nb_cartes, int *cartes, int nb_restantes, int
↪ *restantes, bool action_defausse, ETAT *etat, bool *act_oss_simu)
431 {
432     /*Simule un coup et renvoie son score pour le coup simulé joué*/
433     // INITIALISATION
434     D_FLOAT *res = init_d_float(donnees);
435     int nb_mon_cote = 8;

```

```

436 int nb_cote_adv = 8;
437 int nb_total = nb_cartes + nb_restantes;
438 int *total = malloc(7 * sizeof(int));
439 int *total_s_moi = malloc(7 * sizeof(int));
440 int *cartes_adv = malloc(7 * sizeof(int));
441 int *cartes_moi = malloc(7 * sizeof(int));
442 for (int i = 0; i < 7; i++)
443 {
444     nb_mon_cote -= etat->valide_moi[i];
445     nb_cote_adv -= etat->valide_adv[i];
446     total[i] = cartes[i] + restantes[i];
447     total_s_moi[i] = cartes[i] + restantes[i];
448     cartes_adv[i] = etat->valide_adv[i] + cartes[i] + restantes[i];
449     cartes_moi[i] = etat->valide_moi[i];
450 }
451 marq *mon_cote = init_marqueur(nb_mon_cote, nb_total, total);
452 marq *defausse;
453
454 // Distribution des cartes
455 while (mon_cote->pointeurs != NULL)
456 {
457     if (verification(nb_cartes, cartes, nb_restantes, restantes,
458 ↪ nb_mon_cote, mon_cote->pointeurs, etat, act_poss_simu))
459     {
460         for (int i = 0; i < mon_cote->k; i++)
461         {
462             total_s_moi[mon_cote->pointeurs[i]] -= 1;
463             cartes_adv[mon_cote->pointeurs[i]] -= 1;
464             cartes_moi[mon_cote->pointeurs[i]] += 1;
465         }
466         if (!action_defausse)
467         {
468             defausse = init_marqueur(3, nb_total - mon_cote->k,
469 ↪ total_s_moi);
470         }
471         else
472         {
473             defausse = init_marqueur(5, nb_total - mon_cote->k,
474 ↪ total_s_moi);
475         }
476         while (defausse->pointeurs != NULL)
477         {
478             for (int i = 0; i < defausse->k; i++)
479             {
480                 cartes_adv[defausse->pointeurs[i]] -= 1;
481             }
482             ajout(res, cartes_moi, cartes_adv, g.etat->avantage,
483 ↪ mon_cote->n, mon_cote->k, mon_cote->cartes,
484 ↪ mon_cote->pointeurs, defausse->n, defausse->k,
485 ↪ defausse->cartes, defausse->pointeurs); // On calcule le
486 ↪ score de cette fin de partie
487             for (int i = 0; i < defausse->k; i++)

```

```

481         {
482             cartes_adv[defausse->pointeurs[i]] += 1;
483         }
484         choix_cartes(defausse);
485     }
486
487     // On remet à l'état initial
488     free_marq(defausse);
489     for (int i = 0; i < mon_cote->k; i++)
490     {
491         total_s_moi[mon_cote->pointeurs[i]] += 1;
492         cartes_adv[mon_cote->pointeurs[i]] += 1;
493         cartes_moi[mon_cote->pointeurs[i]] -= 1;
494     }
495 }
496 choix_cartes(mon_cote);
497 }
498
499 // Libération de mémoire
500 free_marq(mon_cote);
501 free(total);
502 free(total_s_moi);
503 free(cartes_adv);
504 free(cartes_moi);
505 return total_simu(res);
506 }

```

3.1.3 Fonction principale jouer_tour

Initialisation

```

508 // Fonction appelée au début du jeu
509 void init_jeu(void)
510 {
511     // ALLOCATION DE MEMOIRE
512     t1 = currenttime();
513     printf("Debut match\n");
514     g.cartes = malloc(7 * sizeof(int));
515     g.etat = malloc(sizeof(ETAT));
516     g.etat->valide_adv = malloc(7 * sizeof(int));
517     g.etat->valide_moi = malloc(7 * sizeof(int));
518     g.etat->avantage = malloc(7 * sizeof(int));
519     g.act_poss = malloc(4 * sizeof(bool));
520     g.restantes = malloc(7 * sizeof(int));
521     moi = id_joueur();
522     if (moi == 0)
523     {
524         adv = 1;
525     }
526     else
527     {
528         adv = 0;
529     }

```



```

530     donnees = creation("stats_cartes_doub.txt");
531     printf("Fin de l'initialisation du tour : %ld\n\n", currenttime() - t1);
532 }
533
534 // Fonction appelée au début du tour
535 void jouer_tour(void)
536 {
537     // INITIALISATION TOUR
538     printf("Debut : manche %d tour %d\n", manche(), tour());
539     t1 = currenttime();
540     update(true);
541     printf("Update termine : 1: %d 2: %d 3 : %d 4: %d en main : %d\n",
542           ↪ g.act_poss[0], g.act_poss[1], g.act_poss[2], g.act_poss[3],
543           ↪ g.en_main);
544     debug_cartes(7, g.cartes, "Mes cartes");
545     debug_cartes(7, g.restantes, "Cartes restantes");
546     float score_maxi = -50;
547     COUP coup_maxi;
548     coup_maxi.action = -1;
549     coup_maxi.cartes = malloc(4 * sizeof(int));
550     coup_maxi.cartes[0] = -1;
551     coup_maxi.cartes[1] = -1;
552     coup_maxi.cartes[2] = -1;
553     coup_maxi.cartes[3] = -1;
554     float res;
555     marq *tour_simu;
556     int *cartes_simu = malloc(7 * sizeof(int));
557     ETAT *etat_simu = malloc(sizeof(ETAT));
558     etat_simu->avantage = malloc(7 * sizeof(int));
559     etat_simu->valide_adv = malloc(7 * sizeof(int));
560     etat_simu->valide_moi = malloc(7 * sizeof(int));
561     bool *act_poss_simu = malloc(4 * sizeof(int));
562     for (int i = 0; i < 4; i++)
563     {
564         act_poss_simu[i] = g.act_poss[i];
565     }
566     for (int c = 0; c < 7; c++)
567     {
568         cartes_simu[c] = g.cartes[c];
569         etat_simu->avantage[c] = g.etat->avantage[c];
570         etat_simu->valide_adv[c] = g.etat->valide_adv[c];
571         etat_simu->valide_moi[c] = g.etat->valide_moi[c];
572     }
573     printf("Fin de copie de l'etat du jeu : %ld\n", currenttime() - t1);
574     fflush(stdout);

```

Valider

```

574     // VALIDER UNE CARTE
575     if (g.act_poss[0])
576     {
577         act_poss_simu[0] = false;
578         tour_simu = init_marqueur(1, g.en_main, g.cartes);

```

```

579     while (tour_simu->pointeurs != NULL)
580     {
581         cartes_simu[tour_simu->pointeurs[0]] -= 1;
582         etat_simu->valide_moi[tour_simu->pointeurs[0]] += 1;
583         res = simulation_coup(tour_simu->n - tour_simu->k, cartes_simu,
584                               ↪ g.nb_restantes, g.restantes, g.act_poss[1], etat_simu,
585                               ↪ act_poss_simu);
586         if (res > score_maxi) // On regarde quel coup à un score maximal
587         {
588             score_maxi = res;
589             coup_maxi.action = 1;
590             coup_maxi.cartes[0] = tour_simu->pointeurs[0];
591         }
592         // APRES LA SIMULATION
593         cartes_simu[tour_simu->pointeurs[0]] += 1;
594         etat_simu->valide_moi[tour_simu->pointeurs[0]] -= 1;
595         choix_cartes(tour_simu);
596     }
597     free_marq(tour_simu);
598     act_poss_simu[0] = true;
599 }
600 printf("Fin simu valider\n");

```

Défausser

```

600     // DEFAUSSER UNE CARTE
601     if (g.act_poss[1])
602     {
603         act_poss_simu[1] = false;
604         tour_simu = init_marqueur(2, g.en_main, g.cartes);
605         while (tour_simu->pointeurs != NULL)
606         {
607             cartes_simu[tour_simu->pointeurs[0]] -= 1;
608             cartes_simu[tour_simu->pointeurs[1]] -= 1;
609             res = simulation_coup(tour_simu->n - tour_simu->k, cartes_simu,
610                                   ↪ g.nb_restantes, g.restantes, false, etat_simu, act_poss_simu);
611             if (res > score_maxi)
612             {
613                 score_maxi = res;
614                 coup_maxi.action = 2;
615                 coup_maxi.cartes[0] = tour_simu->pointeurs[0];
616                 coup_maxi.cartes[1] = tour_simu->pointeurs[1];
617             }
618             // APRES LA SIMULATION
619             cartes_simu[tour_simu->pointeurs[0]] += 1;
620             cartes_simu[tour_simu->pointeurs[1]] += 1;
621             choix_cartes(tour_simu);
622         }
623         free_marq(tour_simu);
624         act_poss_simu[1] = true;
625     }
626     printf("Fin simu defausser\n");

```

Choix trois

```

627 float score_mini;
628 COUP coup_mini;
629 coup_mini.cartes = malloc(4 * sizeof(int));
630 coup_mini.cartes[0] = -1;
631 coup_mini.cartes[1] = -1;
632 coup_mini.cartes[2] = -1;
633 coup_mini.cartes[3] = -1;
634 if (g.act_poss[2])
635 {
636     act_poss_simu[2] = false;
637     score_mini = 50;
638     coup_mini.action = 3;
639     tour_simu = init_marqueur(3, g.en_main, g.cartes);
640     while (tour_simu->pointeurs != NULL)
641     {
642         cartes_simu[tour_simu->pointeurs[0]] -= 1;
643         cartes_simu[tour_simu->pointeurs[1]] -= 1;
644         cartes_simu[tour_simu->pointeurs[2]] -= 1;
645         for (int c = 0; c < 3; c++)
646         {
647             /*On regarde le score en fonction de la carte qu'il va
648             ↪ prendre*/
649             etat_simu->valide_adv[tour_simu->pointeurs[c]] += 1;
650             etat_simu->valide_moi[tour_simu->pointeurs[permu_trois[c][0]]]
651             ↪ += 1;
652             etat_simu->valide_moi[tour_simu->pointeurs[permu_trois[c][1]]]
653             ↪ += 1;
654
655             res = simulation_coup(tour_simu->n - tour_simu->k,
656             ↪ cartes_simu, g.nb_restantes, g.restantes, g.act_poss[1],
657             ↪ etat_simu, act_poss_simu);
658             if (res < score_mini)
659             { // On choisit le score minimisant les coups qu'il fait
660                 score_mini = res;
661                 coup_mini.cartes[0] = tour_simu->pointeurs[0];
662                 coup_mini.cartes[1] = tour_simu->pointeurs[1];
663                 coup_mini.cartes[2] = tour_simu->pointeurs[2];
664             }
665             etat_simu->valide_adv[tour_simu->pointeurs[c]] -= 1;
666             etat_simu->valide_moi[tour_simu->pointeurs[permu_trois[c][0]]]
667             ↪ -= 1;
668             etat_simu->valide_moi[tour_simu->pointeurs[permu_trois[c][1]]]
669             ↪ -= 1;
670         }
671     }
672
673     if (score_mini != 50 && score_mini > score_maxi)
674     { // On choisit le coup qui maximise les 'pires choix possibles'
675         score_maxi = score_mini;
676         coup_maxi.action = coup_mini.action;
677         coup_maxi.cartes[0] = coup_mini.cartes[0];
678         coup_maxi.cartes[1] = coup_mini.cartes[1];

```

```

671         coup_maxi.cartes[2] = coup_mini.cartes[2];
672     }
673     // On revient à l'état initial
674     cartes_simu[tour_simu->pointeurs[0]] += 1;
675     cartes_simu[tour_simu->pointeurs[1]] += 1;
676     cartes_simu[tour_simu->pointeurs[2]] += 1;
677     choix_cartes(tour_simu);
678 }
679 free_marq(tour_simu);
680 act_poss_simu[2] = true;
681 }
682 printf("Fin simu choix 3\n");

```

Choix paquets

```

684 // CHOIX DES PAQUETS
685 int cpt, cpt_adv, place;
686 if (g.act_poss[3])
687 {
688     act_poss_simu[3] = false;
689     coup_mini.action = 4;
690     int *cartes_choisis = malloc(7 * sizeof(int));
691     int *ordre = malloc(4 * sizeof(int));
692     tour_simu = init_marqueur(4, g.en_main, g.cartes);
693     aucune_carte(cartes_choisis);
694     while (tour_simu->pointeurs != NULL)
695     {
696         cartes_simu[tour_simu->pointeurs[0]] -= 1;
697         cartes_simu[tour_simu->pointeurs[1]] -= 1;
698         cartes_simu[tour_simu->pointeurs[2]] -= 1;
699         cartes_simu[tour_simu->pointeurs[3]] -= 1;
700         cartes_choisis[tour_simu->pointeurs[0]] += 1;
701         cartes_choisis[tour_simu->pointeurs[1]] += 1;
702         cartes_choisis[tour_simu->pointeurs[2]] += 1;
703         cartes_choisis[tour_simu->pointeurs[3]] += 1;
704         marq *prem_paquet = init_marqueur(2, 4, cartes_choisis);
705         while (prem_paquet->pointeurs != NULL)
706         {
707             // Pour chaque possibilité d'associer les cartes 2 par 2
708             cpt = 0;
709             cpt_adv = 2;
710             score_mini = 50;
711             // POSSIBILITE 1 : il prend le premier paquet
712             for (int c = 0; c < 7; c++)
713             {
714                 place = 0;
715                 if ((c == prem_paquet->pointeurs[0]))
716                 {
717                     etat_simu->valide_moi[c] += 1;
718                     ordre[cpt] = c;
719                     cpt++;
720                     place++;
721                 }

```

```

722     if (c == prem_paquet->pointeurs[1])
723     {
724         etat_simu->valide_moi[c] += 1;
725         ordre[cpt] = c;
726         cpt++;
727         place++;
728     }
729     for (int tmp = 0; tmp < cartes_choisis[c] - place; tmp++)
730     {
731         etat_simu->valide_adv[c] += 1;
732         ordre[cpt_adv] = c;
733         cpt_adv++;
734     }
735 }
736 if (!(cpt == 2 && cpt_adv == 4))
737 {
738     printf("BOUCLE NON CORRECTE : %d %d", cpt, cpt_adv);
739     fflush(stdout);
740 }
741 assert(cpt == 2 && cpt_adv == 4);
742 res = simulation_coup(tour_simu->n - tour_simu->k,
743     ↪ cartes_simu, g.nb_restantes, g.restantes, g.act_poss[1],
744     ↪ etat_simu, act_poss_simu);
745 if (res < score_mini)
746 {
747     score_mini = res;
748     coup_mini.cartes[0] = ordre[0];
749     coup_mini.cartes[1] = ordre[1];
750     coup_mini.cartes[2] = ordre[2];
751     coup_mini.cartes[3] = ordre[3];
752 }
753
754 etat_simu->valide_moi[ordre[0]] -= 1;
755 etat_simu->valide_moi[ordre[1]] -= 1;
756 etat_simu->valide_adv[ordre[2]] -= 1;
757 etat_simu->valide_adv[ordre[3]] -= 1;
758 // DEUXIEME POSSIBILITE : il prend le deuxième paquet
759 etat_simu->valide_adv[ordre[0]] += 1;
760 etat_simu->valide_adv[ordre[1]] += 1;
761 etat_simu->valide_moi[ordre[2]] += 1;
762 etat_simu->valide_moi[ordre[3]] += 1;
763
764 res = simulation_coup(tour_simu->n - tour_simu->k,
765     ↪ cartes_simu, g.nb_restantes, g.restantes, g.act_poss[1],
766     ↪ etat_simu, act_poss_simu);
767 if (res < score_mini)
768 {
769     score_mini = res;
770     coup_mini.cartes[0] = ordre[2];
771     coup_mini.cartes[1] = ordre[3];
772     coup_mini.cartes[2] = ordre[0];
773     coup_mini.cartes[3] = ordre[1];

```

```

770     }
771
772     etat_simu->valide_adv[ordre[0]] -= 1;
773     etat_simu->valide_adv[ordre[1]] -= 1;
774     etat_simu->valide_moi[ordre[2]] -= 1;
775     etat_simu->valide_moi[ordre[3]] -= 1;
776
777     if (score_mini != 50 && score_mini > score_maxi)
778     {
779         score_maxi = score_mini;
780         coup_maxi.action = coup_mini.action;
781         coup_maxi.cartes[0] = coup_mini.cartes[0];
782         coup_maxi.cartes[1] = coup_mini.cartes[1];
783         coup_maxi.cartes[2] = coup_mini.cartes[2];
784         coup_maxi.cartes[3] = coup_mini.cartes[3];
785     }
786     choix_cartes(prem_paquet);
787 }
788 free_marq(prem_paquet);
789
790 //==== retour à l'état initial
791 cartes_simu[tour_simu->pointeurs[0]] += 1;
792 cartes_simu[tour_simu->pointeurs[1]] += 1;
793 cartes_simu[tour_simu->pointeurs[2]] += 1;
794 cartes_simu[tour_simu->pointeurs[3]] += 1;
795 cartes_choisis[tour_simu->pointeurs[0]] -= 1;
796 cartes_choisis[tour_simu->pointeurs[1]] -= 1;
797 cartes_choisis[tour_simu->pointeurs[2]] -= 1;
798 cartes_choisis[tour_simu->pointeurs[3]] -= 1;
799 choix_cartes(tour_simu);
800 }
801 // désallocation de la mémoire
802 free_marq(tour_simu);
803 free(ordre);
804 free(cartes_choisis);
805 act_poss_simu[3] = true;
806 }

```

Choix de l'action

```

807     printf("FIN SIMU : %d\n", coup_maxi.action);
808     fflush(stdout);
809     // Joue l'action
810     if (coup_maxi.action == 1)
811     {
812         joue_valide(coup_maxi.cartes[0]);
813     }
814     else if (coup_maxi.action == 2)
815     {
816         joue_defausse(coup_maxi.cartes[0], coup_maxi.cartes[1]);
817     }
818     else if (coup_maxi.action == 3)
819     {

```

```

820     joue_trois(coup_maxi.cartes[0], coup_maxi.cartes[1],
821               ↪ coup_maxi.cartes[2]);
822 }
823 else if (coup_maxi.action == 4)
824 {
825     joue_quatre(coup_maxi.cartes[0], coup_maxi.cartes[1],
826               ↪ coup_maxi.cartes[2], coup_maxi.cartes[3]);
827 }
828 else
829 {
830     printf("ERREUR ! AUCUNE ACTION JOUEE :\n");
831 }
832 // Désallocation de la mémoire
833 free(coup_mini.cartes);
834 free(coup_maxi.cartes);
835 free(cartes_simu);
836 free(etat_simu->avantage);
837 free(etat_simu->valide_adv);
838 free(etat_simu->valide_moi);
839 free(etat_simu);
840 free(act_poss_simu);
841 printf("SCORE : %f\nTEMPS : %ld
842       ↪ ms\n\n#####\n\n", score_maxi,
843       ↪ currenttime() - t1);
844 }

```

3.1.4 Répondre à l'action 3

```

842 // Fonction appelée lors du choix entre les trois cartes lors de l'action de
843 // l'adversaire (cf tour_precedent)
844 void repondre_action_choix_trois(void)
845 {
846     // INITIALISATION
847     printf("Repondre choix 3\n");
848     t1 = currenttime();
849     update(false);
850     debug_cartes(7, g.cartes, "Mes cartes");
851     debug_cartes(7, g.restantes, "Cartes restantes");
852     action_jouee tp = tour_precedent();
853     int cartes_3[3] = {tp.c1, tp.c2, tp.c3};
854
855     float score_maxi = -50;
856     int coup_max;
857     float res;
858
859     ETAT *etat_simu = malloc(sizeof(ETAT));
860     int *restantes_simu = malloc(7 * sizeof(int));
861     etat_simu->avantage = malloc(7 * sizeof(int));
862     etat_simu->valide_adv = malloc(7 * sizeof(int));
863     etat_simu->valide_moi = malloc(7 * sizeof(int));
864     for (int c = 0; c < 7; c++)
865     {

```

```

866         restantes_simu[c] = g.restantes[c];
867         etat_simu->avantage[c] = g.etat->avantage[c];
868         etat_simu->valide_adv[c] = g.etat->valide_adv[c];
869         etat_simu->valide_moi[c] = g.etat->valide_moi[c];
870     }
871     for (int c = 0; c < 3; c++)
872     {
873         restantes_simu[cartes_3[c]] -= 1;
874     }
875     debug_cartes(7, restantes_simu, "Restantes_simu");
876
877     // On va simuler si on prend chaque'une des possibilités
878     for (int carte_choisie = 0; carte_choisie < 3; carte_choisie++)
879     {
880         etat_simu->valide_moi[cartes_3[carte_choisie]] += 1;
881         etat_simu->valide_adv[cartes_3[permu_trois[carte_choisie][0]]] += 1;
882         etat_simu->valide_adv[cartes_3[permu_trois[carte_choisie][1]]] += 1;
883         res = simulation_coup(g.en_main, g.cartes, g.nb_restantes - 3,
884                               ↪ restantes_simu, g.act_poss[1], etat_simu, g.act_poss);
885
886         if (res > score_maxi)
887         {
888             score_maxi = res;
889             coup_max = carte_choisie;
890         }
891         etat_simu->valide_moi[cartes_3[carte_choisie]] -= 1;
892         etat_simu->valide_adv[cartes_3[permu_trois[carte_choisie][0]]] -= 1;
893         etat_simu->valide_adv[cartes_3[permu_trois[carte_choisie][1]]] -= 1;
894     }
895     // désallocation de mémoire
896     free(restantes_simu);
897     free(etat_simu->avantage);
898     free(etat_simu->valide_adv);
899     free(etat_simu->valide_moi);
900     free(etat_simu);
901     repondre_choix_trois(coup_max);
902     printf("SCORE : %f\nTEMPS : %ld
903       ↪ ms\n\n#####\n\n", score_maxi,
904       ↪ currenttime() - t1);
905 }

```

3.1.5 Répondre à l'action 4

```

904 // Fonction appelée lors du choix entre deux paquet lors de l'action de
905 // l'adversaire (cf tour_precedent)
906 void repondre_action_choix_paquets(void)
907 {
908     // INITIALISATION
909     printf("Repondre choix paquets\n");
910     t1 = currenttime();
911     update(false);
912     debug_cartes(7, g.cartes, "Mes cartes");

```

```

913 debug_cartes(7, g.restantes, "Cartes restantes");
914 action_jouee tp = tour_precedent();
915 int cartes_4[4] = {tp.c1, tp.c2, tp.c3, tp.c4};
916
917 float score_maxi = -50;
918 int coup_max;
919 float res;
920
921 ETAT *etat_simu = malloc(sizeof(ETAT));
922 int *restantes_simu = malloc(7 * sizeof(int));
923 etat_simu->avantage = malloc(7 * sizeof(int));
924 etat_simu->valide_adv = malloc(7 * sizeof(int));
925 etat_simu->valide_moi = malloc(7 * sizeof(int));
926 for (int c = 0; c < 7; c++)
927 {
928     restantes_simu[c] = g.restantes[c];
929     etat_simu->avantage[c] = g.etat->avantage[c];
930     etat_simu->valide_adv[c] = g.etat->valide_adv[c];
931     etat_simu->valide_moi[c] = g.etat->valide_moi[c];
932 }
933 for (int c = 0; c < 4; c++)
934 {
935     restantes_simu[cartes_4[c]] -= 1;
936 }
937 debug_cartes(7, restantes_simu, "Restantes_simu");
938
939 // On va tester si on prend chaque'un des deux paquets
940 for (int carte_choisie = 0; carte_choisie < 2; carte_choisie++)
941 {
942     etat_simu->valide_moi[cartes_4[permu_paquet[carte_choisie][0]]] += 1;
943     etat_simu->valide_moi[cartes_4[permu_paquet[carte_choisie][1]]] += 1;
944     etat_simu->valide_adv[cartes_4[permu_paquet[carte_choisie][2]]] += 1;
945     etat_simu->valide_adv[cartes_4[permu_paquet[carte_choisie][3]]] += 1;
946     res = simulation_coup(g.en_main, g.cartes, g.nb_restantes - 4,
947         ↳ restantes_simu, g.act_poss[1], etat_simu, g.act_poss);
948
949     if (res > score_maxi)
950     {
951         score_maxi = res;
952         coup_max = carte_choisie;
953     }
954     etat_simu->valide_moi[cartes_4[permu_paquet[carte_choisie][0]]] -= 1;
955     etat_simu->valide_moi[cartes_4[permu_paquet[carte_choisie][1]]] -= 1;
956     etat_simu->valide_adv[cartes_4[permu_paquet[carte_choisie][2]]] -= 1;
957     etat_simu->valide_adv[cartes_4[permu_paquet[carte_choisie][3]]] -= 1;
958 }
959 // désallocation de la mémoire
960 free(restantes_simu);
961 free(etat_simu->avantage);
962 free(etat_simu->valide_adv);
963 free(etat_simu->valide_moi);
964 free(etat_simu);

```

```

964 repondre_choix_paquets(coup_max);
965 printf("SCORE : %f\nTEMPS : %ld
↳ ms\n\n#####\n\n", score_maxi,
↳ currenttime() - t1);
966 }
967
968 // Fonction appelée à la fin du jeu
969 void fin_jeu(void)
970 {
971     // Désallocation de la mémoire
972     free(g.cartes);
973     free(g.etat->valide_adv);
974     free(g.etat->valide_moi);
975     free(g.etat->avantage);
976     free(g.etat);
977     free(g.act_poss);
978     free(g.restantes);
979     printf("Fin\n");
980 }

```

3.2 Calculer le score

3.2.1 L'entête calcul_score.h

```

1 #pragma once
2
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <stdbool.h>
6 #include <assert.h>
7 #include "convertir.h"
8
9 typedef struct double_int
10 {
11     int moi; // Somme
12     int adv; // pondération
13 } D_INT;
14
15 typedef struct double_float
16 {
17     float som;
18     float pond;
19     SIX ***stats;
20 } D_FLOAT;
21
22 bool verif_score_final(int *valide_moi, int *valide_adv);
23
24 D_INT score(int *valide_moi, int *valide_adv, int *avantage);
25
26 int diff_score(int *valide_moi, int *valide_adv, int *avantage);
27
28 D_FLOAT *init_d_float(SIX*** donnees);
29

```

```

30 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);
31
32 float total_simu(D_FLOAT *simu);

```

3.2.2 Calculs avec la moyenne calcul_score.c

```

1  #include "calcul_score.h"
2  #include "convertir.h"
3
4  int valeur_c[7] = {2, 2, 2, 3, 3, 4, 5};
5
6  bool verif_score_final(int *valide_moi, int *valide_adv)
7  {
8      int cpt_moi = 0;
9      int cpt_adv = 0;
10     bool possible = true;
11     for (int i = 0; i < 7; i++)
12     {
13         cpt_moi += valide_moi[i];
14         cpt_adv += valide_adv[i];
15         if (valide_moi[i] > valeur_c[i])
16         {
17             printf("TROP DE CARTES DE MEME VALEUR DE MON COTE POUR LA COULEUR
    ↪ %d : %d!\n", i, valide_moi[i]);
18             possible = false;
19         }
20         if (valide_adv[i] > valeur_c[i])
21         {
22             printf("TROP DE CARTES DE MEME VALEUR DU COTE ADVERSE POUR LA
    ↪ COULEUR %d : %d!\n", i, valide_adv[i]);
23             possible = false;
24         }
25     }
26     if (cpt_moi != 8)
27     {
28         printf("LE COMPTE N'EST PAS BON DE MON COTE : %d \n", cpt_moi);
29         possible = false;
30     }
31     if (cpt_adv != 8)
32     {
33         printf("LE COMPTE N'EST PAS BON DU COTE ADVERSE : %d \n", cpt_adv);
34         possible = false;
35     }
36     return possible;
37 }
38
39 D_INT score(int *valide_moi, int *valide_adv, int *avantage)
40 {
41     bool assertion = verif_score_final(valide_moi, valide_adv);
42     if (!assertion)

```

```

43 {
44     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]);
45     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]);
46     fflush(stdout);
47     assert(false);
48 }
49 D_INT res = {0, 0};
50 for (int i = 0; i < 7; i++)
51 {
52     if (valide_moi[i] > valide_adv[i])
53     {
54         res.moi += valeur_c[i];
55     }
56     else if (valide_adv[i] > valide_moi[i])
57     {
58         res.adv += valeur_c[i];
59     }
60     else if (avantage[i] == 1)
61     {
62         res.moi += valeur_c[i];
63     }
64     else if (avantage[i] == -1)
65     {
66         res.adv += valeur_c[i];
67     }
68 }
69 return res;
70 }
71
72 int diff_score(int *valide_moi, int *valide_adv, int *avantage)
73 {
74     D_INT s = score(valide_moi, valide_adv, avantage);
75     return s.moi - s.adv;
76 }
77
78 D_FLOAT *init_d_float(void)
79 {
80     D_FLOAT *res = malloc(sizeof(D_FLOAT));
81     res->pond = 0;
82     res->som = 0;
83     res->stats = NULL;
84     return res;
85 }
86
87 float ponderation(void)
88 {
89     return 1;
90 }

```

```

91 void ajout(D_FLOAT *simu, int *cartes_moi, int *cartes_adv, int *avantages,
92 ↪ int n_m, int k_m, int *cartes_m, int *choix_m, int n_d, int k_d, int
93 ↪ *cartes_d, int *choix_d)
94 {
95     int sco = diff_score(cartes_moi, cartes_adv, avantages);
96     float p = ponderation();
97     simu->som += sco * p;
98     simu->pond += p;
99 }
100 float total_simu(D_FLOAT *simu)
101 {
102     float res = simu->som / simu->pond;
103     free(simu);
104     return res;
105 }
106
107 int main()
108 {
109     // TESTS
110     int *cm = malloc(7 * sizeof(int));
111     int *ca = malloc(7 * sizeof(int));
112     int *av = malloc(7 * sizeof(int));
113     int cm_i[7] = {2, 2, 2, 2, 0, 0, 0};
114     int ca_i[7] = {0, 0, 0, 1, 1, 3, 3};
115     int av_i[7] = {0, 0, 0, 0, 0, 0, 0};
116     for (int i = 0; i < 7; i++)
117     {
118         cm[i] = cm_i[i];
119         ca[i] = ca_i[i];
120         av[i] = av_i[i];
121     }
122     printf("%d\n", cm, ca, av, cm, ca);
123     D_FLOAT *r = init_d_float();
124     ajout(r, cm, ca, av, cm, ca);
125     printf("%f\n", total_simu(r));
126     free(ca);
127     free(cm);
128     free(av);
129     return 0;
130 }

```

3.2.3 Calculs avec la moyenne pondérée statistique

L'entête de convertir.h

```

1 #pragma once
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <stdbool.h>
6 #include <time.h>

```

```

7 #include <assert.h>
8 #define N_MAX 238
9
10 struct sixuplet
11 {
12     int s;
13     int d;
14     int t;
15     int q4;
16     int q5;
17     int prob;
18 };
19
20 typedef struct sixuplet SIX;
21
22 int lin(int k, int s, int d, int t, int q4, int q5);
23
24 int *delin(int i);
25
26 SIX ***creation(char *nom);
27
28 int proba(SIX ***tab, int n, int k, SIX debut, SIX arrive);
29
30 void free_six(SIX ***s);

```

Contenu de convertir.c

```

1 #include "convertir.h"
2
3 int lin(int k, int s, int d, int t, int q4, int q5)
4 {
5     return 1920 * (k - 1) + 240 * s + 30 * d + 6 * t + 2 * q4 + q5;
6 }
7
8 int *delin(int i)
9 {
10     int *res = malloc(6 * sizeof(int));
11     res[5] = i % 2;
12     i = i / 2;
13     res[4] = i % 3;
14     i = i / 3;
15     res[3] = i % 5;
16     i = i / 5;
17     res[2] = i % 8;
18     i = i / 8;
19     res[1] = i % 8;
20     i = i / 8;
21     res[0] = i;
22     return res;
23 }
24
25 SIX ***creation(char *nom)
26 {

```



```

27     int j;
28     SIX ***res_s = malloc(20 * sizeof(SIX **));
29     assert(res_s != NULL);
30     SIX base = {.s = -1, .d = -1, .t = -1, .q4 = -1, .q5 = -1, .prob = -1};
31     for (int i = 0; i < 20; i++)
32     {
33         printf("%d\n", i);
34         j = (1920 * (i + 1) + 240 * 8 + 30 * 8 + 6 * 5 + 2 * 3 + 2);
35         res_s[i] = malloc(j * sizeof(SIX *));
36         assert(res_s[i] != NULL);
37         for (int k = 0; k < j; k++)
38         {
39             res_s[i][k] = malloc(N_MAX * sizeof(SIX));
40             assert(res_s[i][k] != NULL);
41             for (int l = 0; l < N_MAX; l++)
42             {
43                 res_s[i][k][l] = base;
44             }
45         }
46     }
47     SIX en_place;
48     FILE *fichier = fopen(nom, "r");
49     int nb, nb_f, cpt, ind;
50     int nb_f_max = 0;
51     int nb_lecture = 1;
52     int init[7];
53     int res[6];
54     fscanf(fichier, "%d", &nb);
55     printf("%d\n", nb);
56     for (int i = 0; i < nb; i++)
57     {
58         for (int j = 0; j < 7; j++)
59         {
60             fscanf(fichier, "%d", &init[j]);
61             nb_lecture++;
62         }
63         fscanf(fichier, "%d", &nb_f);
64         nb_lecture++;
65         // printf("i : %d nb_f : %d < %d nb_lect : %d\n", i, nb_f, nb_f_max,
66         //     nb_lecture);
67         if (nb_f > nb_f_max)
68         {
69             nb_f_max = nb_f;
70         }
71         ind = lin(init[1], init[2], init[3], init[4], init[5], init[6]);
72         printf("Init : %d %d %d %d %d %d %d\n", init[0], init[1], init[2],
73             init[3], init[4], init[5], init[6]);
74         printf("Ind : %d\n", ind);
75         cpt = 0;
76         for (int j = 0; j < nb_f; j++)
77         {
78             for (int l = 0; l < 6; l++)

```

```

77         {
78             fscanf(fichier, "%d", &res[l]);
79             nb_lecture++;
80         }
81         en_place.s = res[0];
82         en_place.d = res[1];
83         en_place.t = res[2];
84         en_place.q4 = res[3];
85         en_place.q5 = res[4];
86         en_place.prob = res[5];
87         printf("Res : %d %d %d %d %d %d %d\n", res[0], res[1], res[2],
88             res[3], res[4], res[5], cpt);
89         res_s[init[0] - 1][ind][cpt++] = en_place;
90         printf("OK\n");
91     }
92     return res_s;
93 }
94
95 int proba(SIX ***tab, int n, int k, SIX debut, SIX arrive)
96 {
97     int ind = lin(k, debut.s, debut.d, debut.t, debut.q4, debut.q5);
98     SIX val;
99     for (int i = 0; i < N_MAX; i++)
100     {
101         val = tab[n][ind][i]; // PROBLEME ICI <- IMPOSSIBLE D'ACCEDER AU
102         // TABLEAU (mais pas d'erreurs renvoyés)
103         assert(val.s != -1);
104         if (val.s == arrive.s && val.d == arrive.d && val.t == arrive.t &&
105             val.q4 == arrive.q4 && val.q5 == arrive.q5)
106         {
107             return val.prob;
108         }
109     }
110     assert(false);
111     return 0;
112 }
113
114 void free_six(SIX ***s)
115 {
116     int j;
117     for (int i = 0; i < 20; i++)
118     {
119         j = (1920 * (i + 1) + 240 * 8 + 30 * 8 + 6 * 5 + 2 * 3 + 2);
120         for (int k = 0; k < j; k++)
121         {
122             free(s[i][k]);
123         }
124         free(s[i]);
125     }

```

```

126
127 int main()
128 {
129     // TESTS
130     int t1 = time(NULL);
131     creation("stats_cartes_doub.txt");
132     // creation("stats_nb_doub.txt");
133     int t2 = time(NULL);
134     printf("Temps : %d\n", t2 - t1);
135     return 0;
136 }

```

Calculs avec les statistiques calcul_score.c

```

1  #include "calcul_score.h"
2  #include "convertir.h"
3
4  int valeur_c[7] = {2, 2, 2, 3, 3, 4, 5};
5
6  bool verif_score_final(int *valide_moi, int *valide_adv)
7  {
8      int cpt_moi = 0;
9      int cpt_adv = 0;
10     bool possible = true;
11     for (int i = 0; i < 7; i++)
12     {
13         cpt_moi += valide_moi[i];
14         cpt_adv += valide_adv[i];
15         if (valide_moi[i] > valeur_c[i])
16         {
17             printf("TROP DE CARTES DE MEME VALEUR DE MON COTE POUR LA COULEUR
18             ↪ %d : %d!\n", i, valide_moi[i]);
19             possible = false;
20         }
21         if (valide_adv[i] > valeur_c[i])
22         {
23             printf("TROP DE CARTES DE MEME VALEUR DU COTE ADVERSE POUR LA
24             ↪ COULEUR %d : %d!\n", i, valide_adv[i]);
25             possible = false;
26         }
27     }
28     if (cpt_moi != 8)
29     {
30         printf("LE COMPTE N'EST PAS BON DE MON COTE : %d \n", cpt_moi);
31         possible = false;
32     }
33     if (cpt_adv != 8)
34     {
35         printf("LE COMPTE N'EST PAS BON DU COTE ADVERSE : %d \n", cpt_adv);
36         possible = false;
37     }
38     return possible;
39 }

```

```

38
39 D_INT score(int *valide_moi, int *valide_adv, int *avantage)
40 {
41     bool assertion = verif_score_final(valide_moi, valide_adv);
42     if (!assertion)
43     {
44         printf("Moi %d %d %d %d %d %d %d\n", valide_moi[0], valide_moi[1],
45             ↪ valide_moi[2], valide_moi[3], valide_moi[4], valide_moi[5],
46             ↪ valide_moi[6]);
47         printf("Adv %d %d %d %d %d %d %d\n", valide_adv[0], valide_adv[1],
48             ↪ valide_adv[2], valide_adv[3], valide_adv[4], valide_adv[5],
49             ↪ valide_adv[6]);
50         fflush(stdout);
51         assert(false);
52     }
53     D_INT res = {0, 0};
54     for (int i = 0; i < 7; i++)
55     {
56         if (valide_moi[i] > valide_adv[i])
57         {
58             res.moi += valeur_c[i];
59         }
60         else if (valide_adv[i] > valide_moi[i])
61         {
62             res.adv += valeur_c[i];
63         }
64         else if (avantage[i] == 1)
65         {
66             res.moi += valeur_c[i];
67         }
68         else if (avantage[i] == -1)
69         {
70             res.adv += valeur_c[i];
71         }
72     }
73     return res;
74 }
75
76 int diff_score(int *valide_moi, int *valide_adv, int *avantage)
77 {
78     D_INT s = score(valide_moi, valide_adv, avantage);
79     return s.moi - s.adv;
80 }
81
82 D_FLOAT *init_d_float(SIX** donnees)
83 {
84     D_FLOAT *res = malloc(sizeof(D_FLOAT));
85     res->pond = 0;
86     res->som = 0;
87     res->stats = donnees;
88     return res;
89 }

```

```

86 SIX doublons(int *cartes)
87 {
88     SIX res = {.s = 0, .d = 0, .t = 0, .q4 = 0, .q5 = 0, .prob = 0};
89     for (int i = 0; i < 7; i++)
90     {
91         switch (cartes[i])
92         {
93             case 1:
94                 res.s += 1;
95                 break;
96             case 2:
97                 res.d += 1;
98                 break;
99             case 3:
100                 res.t += 1;
101                 break;
102             case 4:
103                 res.q4 += 1;
104                 break;
105             case 5:
106                 res.q5 += 1;
107                 break;
108             default:
109                 break;
110         }
111     }
112     return res;
113 }
114
115 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)
116 {
117     SIX doub_dep = doublons(cartes_m);
118     SIX doub_fin = doublons(choix_m);
119     int prob = proba(simu->stats, n_m, k_m, doub_dep, doub_fin);
120     SIX doub_dep_d = doublons(cartes_d);
121     SIX doub_fin_d = doublons(choix_d);
122     int prob_d = proba(simu->stats, n_d, k_d, doub_dep_d, doub_fin_d);
123     return prob + prob_d;
124 }
125
126 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)
127 {
128     int sco = diff_score(cartes_moi, cartes_adv, avantages);
129     float p = ponderation(simu, n_m, k_m, cartes_m, choix_m, n_d, k_d,
↵ cartes_d, choix_d);
130     simu->som += sco * p;
131     simu->pond += p;
132 }
133

```

```

134 float total_simu(D_FLOAT *simu)
135 {
136     float res = simu->som / simu->pond;
137     free(simu);
138     return res;
139 }
140

```