BEN LTAIFA Adam
4TC G2

PROJET IAT : SUJET 1

MASTERMIND

Partie I

Question 1 *: Proposer une représentation (un codage) pour toute combinaison de pions*

La solution candidate se présente sous la forme d'un vecteur de couleurs de taille 4. Ou de taille k (nombre de pions) si on souhaite généraliser, mais tout au long de mon travail j'ai gardé l'hypothèse de 4 pions et 8 couleurs.

```
9   #Format de la solution candidate:
10  Solution_candidate = [0, 0, 0, 0]
```

Ensuite on détermine l'ensemble des couleurs qui nous servira d'urne pour la composition de la combinaison. Vous constaterez j'ai fait le choix de représenter les couleurs par des entiers.

```
13  Colors = [1, 2, 3, 4, 5, 6, 7, 8]
14  #Ensemble des couleurs possibles:
15  #1:Rouge #2:Bleu #3:Jaune #4:Vert
16  #5:Noir #6:Orange #7:Violet #8:Gris
```

Ensuite chacune des solution candidates qui formeront la première population seront issu d'un tirage aléatoire.

```
24  def tirage():  # Tirage aletoire dans le set de couleurs pour former un candidat valide
25      Hypothese=[rand.choice(Colors) for i in range(len(Solution_candidate))]
26      return(Hypothese)
```

Question 2 *: Combien de solutions candidates existe-t-il pour des combinaisons de N pions avec k couleurs ?*

Grâce à la logique combinatoire, nous savons que si nous disposons de N pions pouvant chacun prendre k couleurs différentes, nous avons k^N combinaisons possibles.

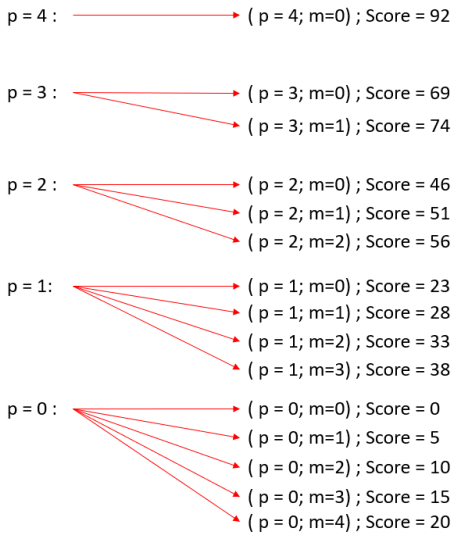
En déduire le nombre de combinaisons pour des combinaisons de 4 pions avec 8 couleurs.

Ainsi avec l'hypothèse suivante, nous savons que nous avons 8^4 combinaisons possibles soit 4096 combinaisons.

Question 3.1 *: Proposer une fonction score (p ∈N, m ∈N) → N + convertissant le nombre de couleurs correctement placées (p) et le nombre de couleurs présentes mais mal placées (m) en un entier positif exprimant un score.*

Ici l'objectif de cette fonction est de mettre des poids sur les indicateurs que nous renvoi la fonction compare () que nous aborderons ensuite. Il semble logique qu'une bonne couleur au bon endroit se voit récompenser par un score plus élevé qu'une bonne couleur au mauvais endroit. Ainsi j'introduit la fonction score de la manière suivante :

```
43  def score(p,m):
44      res = (23*p)+(5*m)
45      return res
```



Cette fonction prendra en paramètres les valeur p et m renvoyées par la fonction compare et nous pondérons ces indicateurs afin d'avoir un score représentant une récompense. J'ai choisi comme pondération des nombres premiers et suffisamment différents afin que le poids d'une couleur correcte bien placée soit plus forte que celle d'une couleur correcte mal placée et qu'il n'y ait pas de chevauchement de score. Pour déterminé préalablement ces valeurs, j'ai fait conjecture que pour chaque couple (p,m), la somme p+m <= 4 et j'ai dressé la liste des scores possibles pour trouver la bonne pondération qui ne donne pas de chevauchement de score.

Question 3.2 *: On veut maintenant être capable d'évaluer la qualité d'une combinaison candidate, notée c, en fonction d'une combinaison déjà jouée cj. Pour évaluer la qualité de c, on va supposer que c'est la combinaison recherchée. Dans ce cas, le score virtuel de cj par rapport à c doit être aussi proche que possible du score déjà obtenu par cj noté scj. En déduire une fonction simple eval(c,cj) → N + calculant la différence entre le score virtuel de cj par rapport à c et le score déjà obtenu pour cj .*

Le rôle de cette fonction eval() va être de calculer les différence entre les deux scores, autrement entre le score déjà obtenu et le score virtuel :

```
def eval_1(Cj, c):
    p1, m1 = compare(Cj, solution)
    p2, m2 = compare(Cj, c)
    res = score(p1, m1)-score(p2, m2)
    return (abs(res))
```

Dans le cas théorique c'est de cette manière que j'implémenterai cette fonction. Elle prendrait en paramètre les deux combinaisons et appellerai la fonction compare pour récupérer les couples (p,m) et faire faire la différence des scores. La combinaison solution est disponible en variable globale.

Mais la fonction que j'ai implémenté réellement dans mon code est légèrement différente car elle est adaptée afin d'aller chercher les scores déjà calculés dans l'historique afin de simuler le fait qu'elle n'a pas accès à la solution.

```
70  def eval(Tentative, candidat, j):
71      p2, m2 = compare(Tentative, candidat)
72      res = Liste_index_score[j] - score(p2, m2)
73      return(abs(res))
```

Tentative est la solution qui a été testé par le gamemaster et qui a retourné un score. Ce score est contenu dans Liste_index_score qui est une simple liste.

En ce qui concerne la fonction compare, même si elle n'est pas demandée dans le rapport, elle reste intéressante à expliquer :

```
28  def compare(coup,ref):
29      copie_ref = ref.copy()
30      p = 0
31      m = 0
32      for i in range(len(coup)):
33          if coup[i]==copie_ref[i]:
34              p = p+1
35              copie_ref[i] = 0 # Pour eviter de compter plusieurs fois une couleur bien placée
36      for i in range(len(coup)):
37          for j in range(len(coup)):
38              if coup[i] == copie_ref[j]:
39                  m = m+1
40                  copie_ref[j] = 0
41      return(p, m)
```

La fonction compare va prendre deux combinaisons de taille 4. Elle va comparer les éléments de la combinaison à tester(coup) avec une combinaison dite référence(réf).

Elle fait un premier tour de boucle pour découvrir toutes les couleurs bien placées (autrement dit incrémenter p) et les mets à zéros pour qu'elles ne soient pas comptées plusieurs fois. Et ensuite on fait un tour pour découvrir s'il y a de bonnes couleurs mais mal placées afin d'incrémenter m)

Question 3.3 *: En déduire la fonction de Fitness qui compare une combinaison candidate c avec l'historique de tous les couples (question, réponse), que l'on cherche à minimiser*

```
88  def fitness(liste):
89      Sum = 0
90      for i in range(len(HISTORIQUE)):
91          Sum = Sum + eval(HISTORIQUE[i], liste, int(i))
92      return Sum
```

La fitness d'un candidat va être la somme de toutes les evals d'un candidat avec les éléments de l'historique

$$Fitness(c) = \sum_{j=0}^{N-1} eval(Cj, c, j) \;\; avec\; N = taille\_Historique$$

```python
def total_fitness(population, Liste_index_fitness): # Fontion qui itère la fonction précédente

    for i in range(len(population)):
        Liste_index_fitness[i][0] = fitness(population[i])

    return Liste_index_fitness
```

Ensuite la fonction vient itérer la fonction Fitness de telle sorte que la Fitness de tous les candidats soit calculée et stockée dans une liste. Il s'agir en réalité d'une liste de liste [[a_0, b_0] , [a_1,b_1], …] ou le premier élément a_i représente la fitness du candidat et bi son indice dans la liste population. Ce découpage à été pensé afin de procéder à la sélection des m meilleurs candidats ensuite.

Partie 2 : Sélection, croisement, mutation

Question 1 *: Proposer un algorithme ou une approche pour la sélection des m "meilleurs" candidats (m < N) afin de constituer la nouvelle génération des solutions candidates*

Une première solution qui est assez lourde et que j'ai implémenté est celle de la sélection de k meilleurs combinaisons possédant les meilleurs fitness :

```python
def selection_des_meilleurs(nb_des_meilleurs, population, listefitness):
    Liste_des_meilleurs = []
    res = 0
    compteur_de_tours = 0
    stop_condition = 0
    #print(len(Liste_des_meilleurs))
    while len(Liste_des_meilleurs) <= nb_des_meilleurs:
        for i in range(len(population)):
            if listefitness[i][0] == res:
                #print("valeur de la fitness =", listefitness[i][0])
                #print("Indice de la combinaison avant cette fitness =", listefitness[i][1])
                Liste_des_meilleurs.append(population[listefitness[i][1]])
            if len(Liste_des_meilleurs) == nb_des_meilleurs:
                stop_condition = 1
                break
        if stop_condition == 1:
            break
        else:
            res = res+1
            compteur_de_tours = compteur_de_tours + 1

    return Liste_des_meilleurs
```

Cette fonction prend en paramètre le nombre de candidat à sélectionner, le tableau contenant tous les candidats et le tableau contenant toutes les fitness des candidats.

Cette méthode itérative va tout simplement commencer par récupérer tous les candidats qui ont une fitness de 0, puis elle sélectionnera ceux à 1 et ainsi de suite jusqu'à ce que le nombre de candidats à sélectionner soit atteint et on place tous ces candidats dans une nouvelle population qui va subir ensuite les différents brassages génétiques.

J'ai également pensé à la solution d'élitisme qui consistait à garder uniquement le candidat qui avait la meilleure fitness et de faire un tirage sur ceux restant pour donner naissance à la nouvelle population mais un problème se posait quand on avait égalité car on ne sait pas lequel nous ferait avancer le mieux dans la partie. J'ai rejeté cette solution car elle me donnait des résultats moins bons que ma première solution. (Il est possible que je ne l'ai pas correctement réalisé d'où les mauvais résultats)

Nous avons également vu en TD qu'il existait une autre solution sous le nom de sélection par échantillonnage qui se base sur une probabilité de sélectionner un candidat en fonction de sa fitness.

Question 2 : Proposer une ou des opérations simples de mutation sur une solution candidate.

La fonction que j'ai créée est assez simple à comprendre. Grace à un taux de mutation qu'on peut régler, on va déterminer le nombre de mutation n qui se produiront. La fonction va choisir n candidat aléatoirement et va leur faire subir des mutations aléatoires par permutation. Autrement dit, pour un candidat donné on va tirer au hasard deux indices et on va échanger les couleurs à ces deux indices précis. Pour que la mutation soit efficace il ne faut pas que la combinaison mutée soit identique à la combinaison originale. J'ai donc dû faire une disjonction de cas.

```
158  def mutation(taux_de_mutation, Liste_des_meilleurs):
159      longueur = len(Liste_des_meilleurs)
160      nb_de_mutation = math.floor(longueur * taux_de_mutation)
161      for i in range(nb_de_mutation):
162          indice_1 = rand.choice([0, 1, 2, 3])
163          indice_2 = rand.choice([0, 1, 2, 3])
164          x = rand.choice([t for t in range(longueur)])
165          etape1 = Liste_des_meilleurs[x].copy()
166          etape2 = Liste_des_meilleurs[x].copy()
167          if indice_1 == indice_2 and indice_2 == 3:
168              indice_2 = indice_2-rand.choice([0, 1, 2])
169              etape1[indice_1] = etape2[indice_2]
170              etape1[indice_2] = etape2[indice_1]
171          elif indice_1 == indice_2 and indice_2 == 0:
172              indice_2 = rand.choice([1, 2, 3]) - indice_2
173              etape1[indice_1] = etape2[indice_2]
174              etape1[indice_2] = etape2[indice_1]
175          elif indice_1 == indice_2:
176              r = [0, 1, 2, 3]
177              r.remove(int(indice_2))
178              indice_2 = rand.choice(r)
179              etape1[indice_1] = etape2[indice_2]
180              etape1[indice_2] = etape2[indice_1]
181          else:
182              etape1[indice_1] = etape2[indice_2]
183              etape1[indice_2] = etape2[indice_1]
184          Liste_des_meilleurs[x] = etape1
185      return (Liste_des_meilleurs)
```

Une autre méthode aurait pu être de tirer au hasard un seul indice et de changer aléatoirement la couleur à cet indice.

***Question 3** : Proposer une ou des opérations de croisement transformant 2 solutions candidates en 1 nouvelle solution candidate. Assurez-vous que les solutions candidates construites sont valides.*

La solution que j'ai choisi pour le croisement est une permutation des branches de deux combinaisons. En effet on va prendre deux combinaisons aléatoires dans les k meilleurs et on va tirer un indice au hasard, qui servira à déterminer à quel indice on divisera chaque combinaison en deux parties. On rassemblera ensuite la partie 1 de la première et la partie 2 de la seconde pour créer un nouveau candidat.

```python
def brassage_genetique(Liste_des_meilleurs):
    iteration = 2*len(Liste_des_meilleurs)
    Temp=[]
    for i in range(iteration):
        cut1 = rand.choice([0, 1, 2])
        part_1 = rand.choice(Liste_des_meilleurs)
        part_2 = rand.choice(Liste_des_meilleurs)

        while part_1 == part_2: #pour faire en sorte qu'on croise pas deux memes combinaisons
            part_2 = rand.choice(Liste_des_meilleurs)

        nouveau_element_1 = part_1[0:(cut1+1)]+part_2[(cut1+1):4]
        Temp.append(nouveau_element_1)

        if len(Liste_des_meilleurs) > 2000:
            break
    Liste_des_meilleurs=Liste_des_meilleurs+Temp
    return Liste_des_meilleurs


def mutation(taux_de_mutation, Liste_des_meilleurs):...
```