Rapport technique - SAÉ 1.01

C. VANDAMME ET K. MASMEJEAN

SOMMAIRE :

Introduction :

Ce projet porte sur la création d'un Mastermind. Le Mastermind est un jeu de réflexion et de déduction inventé par Mordecai Meirowitz (1930-X), un expert israélien en télécommunication, qui se joue à deux. Le but du jeu consiste à trouver une combinaison secrète définie par le premier joueur (codificateur). Le deuxième joueur (décodeur) dispose, dans la version de ce projet, de cinq pions et de quinze lignes (essais), et doit, par déductions successives, deviner la couleur et la position des pions de la combinaison secrète. Le codificateur, afin d'orienter le décodeur, place, sur le côté, un pion blanc par couleur bien placée, et/ou un pion noir par bonne couleur mal placée.

I. Documentation

Ce présent chapitre a pour objectif de présenter la documentation du projet, comme attendu dans les livrables de celui-ci. Ce chapitre sera présenté par fichier, afin de simplifier la lecture.

A. Fichier genSecret.py :

```python
from mm import TabCouleur
from random import *

"""
Fonction qui génère la combinaison secrète

Elle pioche parmi les couleurs définies dans un tableau en générant un nombre aléatoire compris dans les bornes de celui-ci.
"""
def genSecret():
    arr = []

    for i in range(5):
        arr.append(TabCouleur[randint(0, len(TabCouleur) - 1)])

    return arr

if(__name__ == "__main__"):
    for i in range(10):
        print(genSecret())
```

Présentation générale : Ce fichier contient la fonction éponyme **genSecret**, qui, comme son nom l'indique, permet de générer la combinaison secrète que le joueur devra trouver au cours de sa partie.

Fonctionnement : Cette fonction choisit dans un tableau, en l'occurrence le tableau *TabCouleur* de la librairie **mm.py**, et y associe un nombre aléatoire généré grâce à la librairie **random**, notamment grâce à la fonction **randint**. Cette valeur aléatoire permet de choisir une couleur au hasard comprise dans les bornes du tableau *TabCouleur*, et l'ajoute à la variable *arr*. La fonction effectue cette démarche cinq fois, car la combinaison a une taille de cinq couleurs. Enfin, chaque valeur arbitraire est ajoutée à *arr*, puis *arr* est renvoyée.

Variables :

NOM	TYPE	UTILITÉ
arr	list	Contient la combinaison secrète
TabCouleur	list	Contient les couleurs du jeu

Pseudo-code :

```
DÉBUT FONCTION genSecret()
  variable globale :
   TabCouleur:list
  variable locale :
   arr:list
   i:integer
  instructions :
   i ÉGAL 0
   POUR i DE [0;5[ FAIRE :
      index ÉGAL random ENTRE [0;TAILLE(TabCouleur)[
      arr.ajouterFin(TabCouleur[index])
      i ÉGAL i + 1
   FIN POUR
   renvoyer arr
FIN FONCTION genSecret()
```

Jeux de test : Ces jeux de test permettent de vérifier que les tuples renvoyés sont bien aléatoires. Comme attendu, ces derniers sont bel et bien choisis au hasard, ce qui confirme que la fonction **genSecret** renvoie des combinaisons différentes et arbitraires.

```
PS C:\Users\Kevin\Documents\GitHub\SAE_101> & C:/Python312/python.exe c:/Users/
pygame 2.6.1 (SDL 2.28.4, Python 3.12.6)
Hello from the pygame community. https://www.pygame.org/contribute.html
[(128, 128, 128), (255, 0, 0), (0, 0, 255), (255, 0, 127), (255, 255, 255)]
[(128, 128, 128), (128, 128, 128), (255, 0, 127), (255, 0, 127), (0, 0, 255)]
[(255, 0, 127), (0, 255, 0), (0, 0, 255), (0, 0, 255), (0, 255, 0)]
[(255, 0, 0), (225, 127, 0), (255, 255, 255), (0, 0, 255), (0, 255, 0)]
[(0, 255, 0), (0, 0, 0), (255, 0, 127), (0, 0, 255), (255, 0, 127)]
[(255, 255, 255), (225, 127, 0), (255, 255, 255), (0, 255, 0), (225, 127, 0)]
[(0, 0, 0), (0, 0, 255), (225, 127, 0), (255, 0, 0), (225, 127, 0)]
[(128, 128, 128), (128, 128, 128), (0, 255, 0), (255, 255, 255), (0, 0, 255)]
[(255, 255, 255), (0, 0, 0), (0, 255, 0), (0, 0, 0), (225, 127, 0)]
[(255, 255, 255), (128, 128, 128), (0, 0, 255), (0, 0, 255), (255, 0, 127)]
PS C:\Users\Kevin\Documents\GitHub\SAE_101>
```

B. Fichier gameState.py :

```python
"""
Prends en paramètre la combinaison secrète, la dernière tentative de l'utilisateur et le nombre de tentatives.

Retourne si le jeu est fini, et qui est le vainqueur (dans le cas où le jeu n'est pas fini cette valeur n'a pas de sens)
"""
def gameState(secret, line, currentLine):
    # Si le nombre d'essais est supérieur ou égal à 15, le jeu est fini par une défaite
    if(currentLine >= 15):
        return (True, False)

    i = 0

    # On compare chaque élément de la tentative et combinaison secrète
    while(i < len(secret)):
        # Si pour une paire d'éléments, les éléments sont différents, le jeu continue.
        if(secret[i] != line[i]):
            return (False, False)
        i += 1

    # Sinon, le jeu est fini sur une victoire de l'utilisateur
    return (True, True)

if (__name__ == "__main__"):
    def printResult(secret, line, currentLine, result):
        print("gameState(", secret, ",", line, ",", currentLine, ") Attendue:", result, " Retourné:", gameState(secret, line, currentLine))

    # Jeux de test
    printResult([1, 2, 3, 4, 4], [4, 4, 4, 1, 2], 15, (True, False))
    printResult([1, 2, 3, 4, 4], [1, 2, 3, 4, 4], 8, (True, True))
    printResult([1, 2, 3, 4, 4], [4, 4, 3, 2, 1], 8, (False, False))
```

Présentation générale : Ce fichier contient la fonction éponyme **gameState**, qui gère l'état du jeu (en cours, gagné, ou perdu), en particulier la gestion de la comparaison entre la combinaison du joueur et la combinaison secrète.

Fonctionnement : Cette fonction permet de comparer les combinaisons du joueur avec celle de l'ordinateur. Pour ce faire, la fonction utilise une boucle *while* qui effectue une comparaison élément par élément. Si le nombre d'essais du joueur est supérieur ou égal à 15, le joueur a perdu par défaut, sinon, si aucun élément ne diffère dans la comparaison, alors le joueur a gagné. Cette fonction utilise un tuple, composé de deux booléens : le premier définit si le jeu est fini ou non, et le deuxième vaut *True* si le joueur a gagné, sinon le joueur a perdu.

Variables :

NOM	TYPE	UTILITÉ
secret	list	Contient la combinaison secrète
line	list	Contient la combinaison actuelle du joueur
currentLine	integer	Contient le nombre de lignes occupées

Pseudo-code :

```
DÉBUT FONCTION gameState(secret, line, currentLine)
  variable locale :
   i:integer
  instructions :
   i ÉGAL 0
   SI currentLine SUPÉRIEUR OU ÉGAL 15 FAIRE :
      renvoyer Vrai, Faux
   TANT QUE i INFÉRIEUR À LONGUEUR DE secret FAIRE :
      SI i-ième ÉLÉMENT DE secret DIFFÉRENT DE i-ième ÉLÉMENT DE line FAIRE :
         renvoyer Faux, Faux
         i ÉGAL i + 1
   FIN TANT QUE
   renvoyer Vrai, Vrai
FIN FONCTION gameState(secret, line, currentLine)
```

Jeux de test : Ces jeux de test permettent de vérifier que la logique du jeu fonctionne correctement. Ainsi, nous avons retourné la fonction **gameState** avec les valeurs correspondant respectivement à : une partie perdue par manque de coups jouables, une partie gagnée avec la bonne combinaison, puis une partie en cours avec les bonnes couleurs mal placées. Comme attendu, la logique de jeu respecte bien les conditions et renvoie les résultats escomptés.

```
PS C:\Users\Kevin\Documents\GitHub\SAE_101> & C:/Python312/python.exe c:/Users/Kevin/Documents/GitHub/
gameState( [1, 2, 3, 4, 4] , [4, 4, 4, 1, 2] , 15 ) Attendue: (True, False)  Retourné: (True, False)
gameState( [1, 2, 3, 4, 4] , [1, 2, 3, 4, 4] , 8 ) Attendue: (True, True)  Retourné: (True, True)
gameState( [1, 2, 3, 4, 4] , [4, 4, 3, 2, 1] , 8 ) Attendue: (False, False)  Retourné: (False, False)
```

C. Fichier feedback.py :

```python
"""
Prends en paramètre la combinaison secrète et la tentative de l'utilisateur

Retourne le nombre d'éléments bien placés, et le nombre de couleurs
"""
def feedback(secret, comb):
    i = 0

    wellPlaced = 0 # Compteur pour les couleurs bien placées

    secretDic = {}
    combList = []

    # Parcourt les deux listes (secret et comb) pour comparer les couleurs
    while(i < len(secret)):
        secretColor = secret[i] # Couleur actuelle de la combinaison secrète
        combColor = comb[i]     # Couleur actuelle de la tentative

        # Si les couleurs sont identiques, elles sont bien placées
        if(secretColor == combColor):
            wellPlaced += 1
        # Sinon, on les stocke pour la vérification des "bonnes couleurs mal placées"
        else:
            if(secretColor in secretDic):
                secretDic[secretColor] += 1
            else:
                secretDic[secretColor] = 1
            combList.append(combColor)

        i += 1

    goodColor = 0 # Compteur pour les bonnes couleurs mal placées

    # Parcours des éléments mal placés
    for element in combList:
        # Si l'élément se trouve également dans la combinaison secrète et qu'il n'a pas été compté pour "WellPlaced" ou "goodColor"
        if(element in secretDic and secretDic[element] > 0):
            goodColor += 1          # On incrémente goodColor
            secretDic[element] -= 1 # On décrémente l'élément pour ne pas le recompter

    return (wellPlaced, goodColor)


if (__name__ == "__main__"):
    def printResult(secret, comb, result):
        print("feedback(", secret, ",", comb, ") Attendue:", result, " Retourné:", feedback(secret, comb))

    # Jeux de test
    printResult([1, 2, 3, 4, 4], [4, 4, 4, 1, 2], (0, 4))
    printResult([1, 2, 3, 4, 4], [1, 2, 3, 4, 4], (5, 0))
    printResult([1, 2, 3, 4, 4], [4, 4, 3, 2, 1], (1, 4))
```

Présentation générale : Ce fichier contient la fonction éponyme **feedback**, qui, comme son nom l'indique, sert à retourner les valeurs du jeu pour les couleurs bien placées, et des bonnes couleurs mal placées. Ce fichier, à l'instar de **gameState**, est vital pour le bon déroulement du jeu.

Fonctionnement : La fonction **feedback** admet deux paramètres, qui sont, respectivement, la combinaison secrète et la combinaison actuelle du joueur. Pour ce faire, la fonction utilise une boucle *while* qui permet de comparer les deux combinaisons, et si les couleurs correspondent, alors la variable *wellPlaced* est incrémentée de un. Dans le cas contraire, le compteur de la couleur courante de *secret* est incrémenté de un, et la couleur de la proposition du joueur est ajoutée à *combList*. Par la suite, les éléments de *combList* sont soumis à une comparaison, et si l'élément actuel se trouve dans ce dictionnaire, alors la fonction vérifie que le compteur associé est supérieur à 0, et si c'est le cas on décrémente ce compteur et on incrémente *goodColor*, sinon on ne fait rien. Enfin, la fonction renvoie les valeurs de *wellPlaced* et de *goodColor*.

Variables :

NOM	TYPE	UTILITÉ
wellPlaced	integer	Contient le nombre de couleurs bien placées
goodColor	integer	Contient le nombre de bonnes couleurs mal placées
i	integer	Sert à parcourir les combinaisons
combList	list	Contient les couleurs de la combinaison du joueur, , dans le cas où celles-ci diffèrent de la combinaison secrète au même index
secretColor	tuple	Contient la couleur courante pendant la comparaison de la combinaison secrète avec celle du joueur
combColor	tuple	Contient la couleur courante pendant la comparaison de la combinaison du joueur avec celle de l'ordinateur
secretDic	dict	Contient des compteurs, où les clés sont des couleurs, qui représentent le nombre de fois où la couleur associée apparaît dans la combinaison secrète, **en étant mal placée**

Pseudo-code :

```
DÉBUT FONCTION feedback(secret, comb)
  variables locales :
   wellPlaced:integer
   goodColor:integer
   i:integer
   combList:list
   secretColor:list
   combColor:list
   secretDic:dict
  instructions :
   i ÉGAL 0
   wellPlaced ÉGAL 0
   TANT QUE i INFÉRIEUR À LONGUEUR DE secret FAIRE:
      secretColor ÉGAL secret[i]
      combColor ÉGAL comb[i]
      SI secretColor ÉGAL combColor FAIRE :
         wellPLaced ÉGAL wellPLaced + 1
      SINON FAIRE :
         SI secretColor DANS secretDic FAIRE :
            secretDic[secretColor] ÉGAL secretDic[secretColor] + 1
         SINON FAIRE :
            secretDic[secretColor] ÉGAL 1
         FIN SI
         combList.ajouterFin(combColor)
      FIN SI
      i ÉGAL i + 1
   FIN TANT QUE
   goodColor ÉGAL 0
   POUR element DANS combList FAIRE :
      SI element DANS secretDic ET secretDic[element] SUPÉRIEUR À 0 FAIRE :
         goodColor ÉGAL goodColor + 1
         secretDic[element] ÉGAL secretDic[element] - 1
      FIN SI
   FIN POUR
   renvoyer wellPlaced, goodColor
FIN FONCTION feedback(secret, comb)
```

Jeux de test : Ces jeux de test permettent de vérifier que la fonction **feedback** renvoie les valeurs attendues, en envoyant plusieurs valeurs correspondant à des coups qu'un joueur pourrait essayer. Le premier test renvoie un feedback correspondant à 4 bonnes couleurs mal placées et aucune bonne couleur. Le deuxième correspond à une victoire du joueur, avec 5 couleurs bien placées. Enfin, le dernier test renvoie les valeurs attendues pour une couleur bien placée, et 4 bonnes couleurs mal placées. Avec ces jeux de test, nous savons désormais que la fonction **feedback** fonctionne correctement.

```
PS C:\Users\Kevin\Documents\GitHub\SAE_101> & C:/Python312/python.exe c:/Users/Kevin
feedback( [1, 2, 3, 4, 4] , [4, 4, 4, 1, 2] ) Attendue: (0, 4)  Retourné: (0, 4)
feedback( [1, 2, 3, 4, 4] , [1, 2, 3, 4, 4] ) Attendue: (5, 0)  Retourné: (5, 0)
feedback( [1, 2, 3, 4, 4] , [4, 4, 3, 2, 1] ) Attendue: (1, 4)  Retourné: (1, 4)
```

D. Fonction distance :

```
def distance(a:list,b:list)->float:
    return math.sqrt((a[0]-b[0])**2+(a[1]-b[1])**2)
```

Cette fonction permet de calculer la distance entre deux listes de points donnés. En réalité, cette fonction correspond à une distance euclidienne. En considérant que *a*[0], *a*[1], *b*[0], et *b*[1] sont les coordonnées cartésienne de *a* et *b*, alors la distance euclidienne de *a* et de *b* est facilement calculable à l'aide de la formule suivante :

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}.$$

II. Répartition des tâches

Ce présent chapitre a pour objectif de présenter la répartition des tâches au sein du binôme. De manière générale, M. MASMEJEAN s'est occupé en principal de la rédaction du code, avec l'aide et l'avis de M. VANDAMME. Le projet ayant été conclu rapidement par le binôme, la répartition des tâches au niveau informatique n'a malheureusement pas pu être prise en compte de manière effective.
M. VANDAMME, de son côté, s'est occupé en principal de la rédaction de ce rapport technique et de la réalisation du diaporama, avec l'aide et l'avis de M. MASMEJEAN.
Bien que des efforts aient été fait pour répartir au mieux les tâches, le binôme a promptement conclu ce projet, et la répartition des tâches s'en est vue affectée, bien que les coéquipiers aient participé autant d'un côté que de l'autre.

Conclusion :

Ce projet nous a permis d'explorer la programmation graphique dans un premier temps, et l'écriture de ce rapport technique nous a également permis d'avoir une première expérience dans la rédaction de documentation technique d'un projet, la rédaction de documentation de fonctions informatiques, et la gestion de projet en équipe.

Ce projet fut rapidement conclu en équipe, et sans rencontrer de réelles difficultés autre que la répartition des tâches. Concrètement, ce projet peut-être amélioré de nombreuses façons, comme, par exemple, une interface homme-machine plus agréable et plus moderne, l'ajout de niveaux de difficulté, d'une option ordinateur contre ordinateur, et ainsi de suite. Ce projet constitue une excellente base, et montre que la seule réelle limite à ce projet est notre imagination.

K. MASMEJEAN
C. VANDAMME