La réussite des alliances

par Léo Peyronnet et Gabriel Caux : Binome numéro 3

Février-Avril 2022

Table des matières

1 La réussite des alliances : structuration du jeu

1.1 Rappel des règles du jeu

Quelques rappels des règles :
Prenez un paquet de cartes (32 ou 52) mélangez le pour créer une pioche, ensuite tirez les cartes une par une en les posant de gauche à droite faces visibles. Une fois que trois cartes ont été posées vous pouvez commencer à jouer. On procédera ainsi, on suppose que la carte la plus à gauche est la numéro une, la carte à sa droite la numéro deux, celle à sa droite numéro trois et ainsi de suite. Si la carte numéro trois a une similarité (même couleur ou même valeur) avec la carte numéro une il y a alors alliance de cartes, vous pouvez alors faire un saut : la carte qui est située entre nos deux cartes passe sur le tas de celle qui la précède. Si aucun saut n'est possible il faudra continuer à piocher les cartes jusqu'à ce qu'un ou plusieurs saut soit possible. Le jeu s'arrête lorsqu'il n'est plus possible de piocher de cartes et qu'il n'est pas possible de faire de saut non plus. On compte alors le nombre de tas restant et selon le nombre de tas requis pour gagner vous savez si oui ou non vous avez réussi.

1.2 Création des fonctions (partie guidée)

Pour représenter informatiquement le jeu de la réussite des alliances, nous avions à notre disposition une partie guidée intégrée au sujet. Ce "cahier des charges" détaillé nous a fourni un certain confort quant à l'architecture du programme en lui même. En effet, comme les entrées, sorties et effets de bords de nos fonctions nous étaient déjà indiqué, nous n'avions pas à réfléchir à comment nos fonctions allaient interagir entre elles. Les appels à des fonctions antérieures, s'il y en avait, nous étaient eux aussi indiqués ou tout du moins suggérés. Cette souplesse de travail nous a permis de nous concentrer sur l'algorithmique de nos fonctions et comment optimiser ces dernières.

1.2.1 Fonctions "une_etape_reussite" et "piocher"

Néanmoins, le sujet nous permettait également de prendre certaines libertés. Par exemple, il nous était suggéré de réaliser des fonctions auxiliaires pour programmer la fonction "*une_etape_reussite*". C'est ce que nous avons fait avec la fonction "*piocher*" qui compartimente l'action de piocher une carte. Elle prends en argument deux listes, une qui représente la liste des tas de la réussite, l'autre qui représente la pioche. Elle a pour effet de bord de déplacer la première carte de la pioche (c'est à dire d'indice 0) au dernier emplacement de la liste des tas. Pour cela, nous avons utilisé la fonction "*pop*" qui a elle même pour effet de bord de supprimer un élément d'une liste à un index précis et qui retourne cet élément. Nous avons donc redirigé la sortie de la fonction "*pop*" comme ceci :

```
1       def piocher(liste_tas,pioche):
2               liste_tas+=[pioche.pop(0)]
```

Fonction "piocher" : ligne 133 de fonctions.py

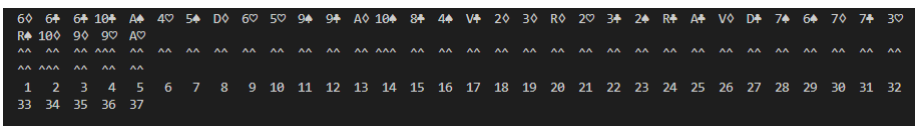
Nous n'avons cependant pas jugé nécessaire de faire d'autres compartimentations pour fluidifier la fonction "*une_etape_reussite*".

1.2.2 Fonctions "reussite_mode_manuel" et "afficher_reussite_num"

La fonction "*reussite_mode_manuel*" nous a permis à nouveau de pouvoir prendre des libertés cette fois-ci quant à l'affichage des éléments et plus globalement l'interface utilisateur. Pour rappel, cette fonction a pour but de laisser l'utilisateur jouer une partie. Ceci nécessite une interface pour que l'utilisateur puisse choisir ce qu'il veut faire. Ainsi, afin de présenter au mieux ses options de jeu à l'utilisateur, nous avons réaliser un menu contenant un choix pour piocher, un choix pour réaliser un saut et un choix pour mettre fin à la partie. En plus de ce menu, nous avons décidé de réaliser un affichage de la réussite plus poussé permettant de relier chaque carte à un nombre et ainsi faciliter la sélection de la carte à faire sauter. Pour cela, nous avons réaliser la fonction "*afficher_reussite_num*". Elle prends pour argument la liste de cartes à afficher, ne renvoie rien et a pour effet de bord l'affichage détaillé de la réussite. Cette fonction consiste en 3 boucles affichant respectivement les cartes de la réussite, les accents circonflexes permettant de souligner les cartes, et les nombres désignant les cartes. A noter que bien que semblant être des index, ces nombres n'en sont pas tout à fait car : soit une liste li, u l'index d'un élément de cette liste et v un des nombres utilisés dans la fonction , alors :

$$u \in [0; (len(li) - 1)] \iff [0; len(li)[$$
$$v \in [1; len(li)]$$

Lors de la création de cette fonction, nous avons constaté des bugs survenant lors de l'affichage de grandes listes de cartes comme une pioche entière d'un jeu de 52 cartes. A raison de 4 caractères imprimés par carte (les 3 caractères de la carte + l'espace de fin du print), l'affichage entier d'un jeu de 52 cartes correspond à l'impression de $4 \times 52 = 208$ caractères. Cela peut être un problème si l'utilisateur possède un petit écran et par conséquent un petit terminal. Les trois lignes de 208 caractères peuvent "déborder" chacune leur tour, provoquant des bugs similaires à celui-ci :



Bug d'affichage relatif à la fonction "afficher_reussite_num" telle qu'écrite au commit 728144c (cf.??)

Nous avons résolu ce problème en imbriquant les trois boucles d'affichage dans une plus grande boucle, et en changeant la condition de la première boucle. Maintenant, la première boucle conditionne les deux suivantes puisqu'elles sont des boucles "for" tournant sur une liste (li_dix) ayant un nombre d'éléments équivalent au nombre de passage dans la première boucle. Cette boucle s'arrête soit lorsqu'il n'y a plus de cartes dans la liste, soit si l'affichage dépasse la taille du terminal.

Le compteur "y" est incrémenté à chaque passage dans la boucle, c'est à dire à chaque impression de carte. Il permet donc de savoir où nous en sommes dans la lecture/impression de la réussite et par conséquent, poser une condition de sortie de boucle lorsque qu'on arrive au bout de la liste de cartes. Le compteur "i" qui est incrémenté lui aussi à chaque impression mais réinitialisé à chaque passage dans la grande boucle (ce qui reviens à une réinitialisation à la sortie de la "petite" boucle).

Donc, multiplié par 4, il permet de compter le nombre de caractères imprimés à chaque passage de grande boucle. Il faut maintenant connaître la largeur du terminal pour pouvoir la comparer à "i". Pour obtenir la taille du terminal, nous avons utilisé le module intégré à python : "*shutil*". Avec la fonction "*get_terminal_size*", on peut obtenir le nombre de caractères que peut contenir le terminal en longueur. Ainsi, nous pouvons construire la deuxième condition de sortie de boucle en spécifiant que la boucle est valide tant que $i \times 4$ est inférieur ou égal à la largeur du terminal. Après modifications, la fonction est donc comme suit :

```
1  def afficher_reussite_num(liCartes):
2      columns, rows = shutil.get_terminal_size()
3      i=0
4      y=0
5      li_dix=[]
6      cpt=1
7      while y<len(liCartes):
8          i=1
9          li_dix=[]
10         while i*4<=columns and y<len(liCartes):
11             carte=liCartes[y]
12             print(carte_to_chaine(carte),end=" ")
13             if carte["valeur"]==10:
14                 li_dix+=[True]
15             else:
16                 li_dix+=[False]
17             y+=1
18             i+=1
19         print()
20         for dix in li_dix:
21             if dix:
22                 print("^"*3,sep="",end=" ")
23             else:
24                 print(" ","^"*2,sep="",end=" ")
25         print()
26         for dix in li_dix:
27             if cpt<10:
28                 print(" "*2,cpt,sep="",end=" ")
29             else:
30                 print(" ",cpt,sep="",end=" ")
31             cpt+=1
32         print("\n")
```

Fonction "afficher_reussite_num" : ligne 24 de fonctions.py

1.2.3 Fonction "texte_encadre"

Finalement, nous avons réalisé la fonction "*texte_encadre*" qui permet d'encadrer un texte pour le mettre en valeur. Nous avons paramétré cette fonction avec deux modes d'encadrement : le premier pour le texte "normal" et l'autre pour les titres. Ainsi, cette fonction prend en arguments une chaine de caractère "texte" correspondante au texte à afficher et un booléen "titre" optionnel correspondant au mode d'affichage. Il est donc de base sur False, ce qui correspond au mode "normal". Comme pour "*afficher_reussite_num*" vu précédement (cf.**??**), cette fonction utilise

le module "shutil" et sa fonction "*get_terminal_size*". Pour rappel, elle permet d'obtenir deux entiers correspondant respectivement à la largeur et à la hauteur du terminal. L'utilisation de cette fonction nous permettra donc d'obtenir la largeur du terminal pour que l'encadrement prenne bien toute la longueur de la ou les lignes qu'il occupera. Après une syntaxe conditionnelle vérifiant la valeur de l'argument "titre", le code de cette fonction correspond à un enchainement de boucles "for" parcourant une range variable. Pour les textes normaux, l'affichage se fait sur une ligne tandis que pour les titres, l'affichage se fait sur trois lignes. La range de la boucle "for" variera en fonction de la ligne sur laquelle elle se trouve.

```
1  def texte_encadre(texte,titre=False):
2      columns, rows = shutil.get_terminal_size()
3      columns-=len(texte)+2 # le +2 pour un espace de chaque côté du texte
4      columns=columns//2
5      if titre:
6          for i in range((columns*2)+len(texte)+2):
7              print("=",end="")
8          print()
9          for i in range(columns):
10             print("=",end="")
11         print(" "+texte+" ",end="")
12         for i in range(columns):
13             print("=",end="")
14         print()
15         for i in range((columns*2)+len(texte)+2):
16             print("=",end="")
17         print()
18     else:
19         for i in range(columns):
20             print("=",end="")
21         print(" "+texte+" ",end="")
22         for i in range(columns):
23             print("=",end="")
24         print()
```

Fonction "texte_encadre" : ligne 56 de fonctions.py

2 Les Extensions : ajouts de fonctionnalités.

Dans une seconde partie du sujet, il nous a été demandé de réaliser des fonctions supplémentaires à celles décrites dans la partie guidée.

2.1 Vérification de la pioche

Parmi les idées d'extensions que propose le sujet, nous avons choisi la fonction permettant de vérifier que la pioche de cartes ne soit pas truquée. Nommée "*vérifier_pioche*", elle prend en arguments la liste que l'on souhaite vérifier ainsi que le nombre de cartes qu'elle contient - ou tout du moins qu'elle est censée contenir ! Elle n'a pas d'effets de bord et renvoie un booléen pour statuer de la conformité de la pioche : True si la pioche est conforme et False si la pioche est truquée. Notre algorithme se base sur la comparaison de la liste passée en argument avec une liste conforme. Une pioche peut être truquée de différentes manières :

- on peut ajouter les cartes que l'on souhaite directement dans la liste, auquel cas la taille de la liste doit être trop grande.
- on peut sinon placer les cartes que l'on souhaite à l'index d'une autre carte. On remplace donc une carte par une autre, ce qui reviens à réécrire les cartes pour qu'elles nous arrangent. Il faudra donc veiller à ce qu'il n'y est pas de double. Pour cela, et comme nous savons que la taille de la liste sera contrôlée ensuite, il nous suffit juste de veiller à ce que toutes les cartes présentes dans une pioche conventionnelle le soient aussi dans la pioche que l'on contrôle. Ainsi, s'il y a des doubles, cette dernière sera forcément d'une taille non conforme.

Nous avons donc fait appel à notre fonction "*init_pioche_alea*" crée lors de la partie guidée qui permet de générer une pioche de 32 ou 52 cartes mélangées aléatoirement. Cette pioche nous servira de comparaison. Nous passons tout d'abord en revue la pioche suspecte en veillant à ce que chaque carte de la pioche conforme soit dans la pioche suspecte. Si ce n'est pas le cas, la fonction renvoie False. La fonction procède ensuite à une vérification de la longueur de la pioche suspecte. Pareil, si la pioche a plus de cartes qu'elle ne devrait en avoir, la fonction renvoie False. Si la pioche possède toutes les cartes de la pioche conforme tout en ayant le bon nombre de cartes, alors la fonction renvoie True.

```
def verifier_pioche(liCartes,nb_cartes=32):
    liCartesVerif=init_pioche_alea(nb_cartes)
    for carteV in liCartesVerif:
        if not(carteV in liCartes):
            return False
    if len(liCartes)!=nb_cartes:
        return False
    return True
```

Fonction "verifier_pioche" : ligne 220 de fonctions.py

2.2 Statistiques

Nous avons aussi choisi de réaliser deux fonctions liées aux statistiques :
La première étant nommée "*res_multi_simulation*" qui va comme son nom l'indique nous servir à faire de multiples simulations de parties en même temps. Elle prend en argument le nombre de simulation que nous voulons faire ainsi que le nombre de carte que notre jeu contient. Voilà comment cette fonction procède, elle crée d'abord une boucle qui va faire appel à deux fonctions de la partie guidée, "*init_pioche_alea*" qui comme décrite précédemment génère une pioche de 32 ou 52 cartes mélangées aléatoirement ainsi que "*reussite_mode_auto*" qui permet de jouer une partie de réussite des alliances automatiquement. Ces deux fonctions nous permettent de faire des simulations différentes à chaque boucle. Après avoir fait chaque simulation de partie, "*res_multi_simulation*" rajoute le résultat dans une liste qui est finalement renvoyée à la fin de la fonction. Cette liste qui est renvoyée correspond donc aux nombres de tas restants pour chaque simulation.

La seconde fonction se nomme "*statistiques_nb_tas*". Elle va nous permettre de connaitre certaines statistiques. Elle prend en argument le nombre de simulation que nous souhaitons faire ainsi que le nombre de carte contenu dans le jeu (32 ou 52). Cette fonction a pour but de faire un nombre défini de simulation puis de trouver ou calculer la moyenne, le maximum ainsi que le minimum de tas obtenus. Pour faire cela elle va d'abord faire appel à la fonction "*res_multi_simulation*" vu précédemment. Nous allons créer une boucle qui va nous servir à trois choses : premièrement grâce à cette boucle nous allons pouvoir additionner toutes les valeurs de la liste obtenue par l'appel de "*res_multi_simulation*" dans une seule variable nommée "moyenne". Nous allons aussi pouvoir trouver le maximum et le minimum en comparant les valeurs de la liste entre elles.

Une fois sorti de cette boucle nous pouvons alors diviser notre variable "moyenne" par la longueur de la liste pour trouver la moyenne. Cette fonction nous affiche donc la moyenne, le maximum et le minimum de tas pour un nombre prédéfinis de simulations. Grâce à cette fonction nous avons pu voir que pour dix milles simulations il y a 21.133 tas en moyenne 2 au minimum et 47 au maximum.

```
def res_multi_simulation(nb_sim,nb_cartes=32):
    result=[]
    for i in range(nb_sim):
        pioche=init_pioche_alea(nb_cartes)
        liCartes=reussite_mode_auto(pioche)
        result+=[len(liCartes)]
    return result
def statistiques_nb_tas(nb_sim,nb_cartes=32):
    appel = res_multi_simulation(nb_sim,nb_cartes)
```

```
10      if nb_sim<=0:
11          print("Pas de valeur dans la liste")
12          return None
13      moyenne = 0
14      maxi = None
15      mini = None
16      for i in range(len(appel)):
17          moyenne += appel[i]
18          if maxi == None or appel[i] > maxi:
19              maxi = appel[i]
20          if mini == None or appel[i] < mini:
21              mini = appel[i]
22      moyenne = moyenne/nb_sim
23      print("La moyenne est :",moyenne)
24      print("le minimum est :",mini)
25      print("le maximum est :",maxi)
```

Fonctions "res_multi_simulation" et "statistiques_nb_tas" : ligne 229 à 253 de fonctions.py

2.3 Probabilités

Pour finaliser nos extensions, nous avons décidé de compléter notre partie "Statistiques" avec une partie "Probabilités". Nous avons réalisé dans un premier temps la fonction "*proba*" qui nous permet de calculer les pourcentages de réussite en fonction du nombre de tas de cartes maximum pour gagner (nombre palier de réussite). Cette fonction ne prend comme argument optionnel que le nombre de cartes présentes dans le jeu (32 ou 52), n'a pas d'effets de bord et retourne une liste contenant les pourcentages de réussite triés par ordre croissant du nombre palier de réussite. L'appel à la fonction précédemment présentée "*res_multi_simulation*"(cf.??) constitue la base de notre algorithme. Pour rappel, elle permet d'obtenir une liste d'entiers correspondants aux nombres de tas restant respectivement à la fin de chaque réussite simulée. Donc, si nous réalisons cent simulations, nous pouvons compter combien respectent le palier de réussite et donc établir un pourcentage de victoire avec tel palier de réussite.

Ainsi, une boucle "for" se charge de vérifier si chaque valeur de la liste est inférieure ou égale au nombre palier. Si oui, une variable "*prctage*" correspondante au pourcentage est incrémentée. Au bout des cent passages de boucle, la valeur de cette variable est ajoutée à la liste de retour puis est remise à zéro. Cette remise à zéro permet l'itération des étapes décrites dans une grande boucle "while" ayant pour condition de sortie qu'un compteur soit supérieur au nombre de cartes du jeu. Ce compteur se nomme "i" et représente le nombre palier pour la réussite. Ainsi, en l'incrémentant à la fin de chaque passage dans sa boucle on peut établir le pourcentage de réussite pour les différentes valeurs de palier. Nous pouvons représenter notre fonction comme suit : Soit i le compteur du même nom, y le compteur permettant de parcourir la liste retourné par "*res_multi_simulation*" [1], *li* la liste que retourne "*proba*", et c le nombre de cartes du jeu [2] (32 ou 52), alors :

Nombre palier de réussite	2	i	c
Valeur de la liste retournée par "*proba*"	li[0]	li[y]	li[$c-2$]

Donc :

$$i \in [2; c]$$
$$y \in [0; c-2]$$

Nous disposons maintenant de pourcentages décrivant les probabilités de réussite des alliances en fonction d'un certain nombre de tas restant à la fin de la partie. Nous avons créé une seconde

1. y n'est pas présent dans le code mais est algorithmiquement sous-entendu par la boucle "for".
2. c est représenté par l'argument optionnel "*nb_cartes*" dans le code de "*proba*".

fonction nommée "*affiche_proba*" afin de représenter graphiquement les probabilités de victoire. Comme la fonction "*proba*", "*affiche_proba*" a pour seul argument un entier indiquant le nombre de cartes présentes dans le jeu. Cependant, elle ne renvoie rien et a pour effet de bord l'affichage du graphique. Nous avons fait appel dans un premier temps à notre fonction "*proba*" afin de remplir notre axe des ordonnées. Les valeurs en abscisses sont générées par une boucle "for". Nous pouvons exprimer x et y, respectivement pour les abscisses et les ordonnées comme ceci, avec c le nombre de cartes du jeu :

$$x \in [2; c]$$
$$y \in [0; 100]$$

Pour l'affichage, nous avons fait appel au module python "*matplotlib.pyplot*" donnant accès à de nombreuses fonctions d'affichage graphique. Pour notre part, nous avons dans un premier temps utilisé la fonction "*plot*" permettant de créer le graphique en lui passant en argument les listes représentant les différents axes du graphique. Pour rendre le graphique plus compréhensible, nous avons également utilisé les fonctions "*xlabel*" et "*ylabel*" permettant de donner un titre aux abscisses et aux ordonnées. Pour finir, afin d'afficher le graphique, nous avons utilisé la fonction "*show*".

```
1  def proba(nb_cartes=32):
2      i=2
3      prctage=0
4      result=[]
5      while i<=nb_cartes:
6          li=res_multi_simulation(100,nb_cartes)
7          for e in li:
8              if e<=i:
9                  prctage+=1
10         result+=[prctage]
11         i+=1
12         prctage=0
13     return result
14 def affiche_proba(nb_cartes=32):
15     x=[]
16     y=proba(nb_cartes)
17     for e in range(2,nb_cartes+1):
18         x+=[str(e)]
19     plt.plot(x,y)
20     plt.xlabel("Nombres de cartes limite pour la victoire")
21     plt.ylabel("Pourcentage de réussite")
22     plt.show()
```

Fonctions "proba" et "affiche_proba" : ligne 255 à 276 de fonctions.py

3 Le debug_mode

Nous sommes arrivés à un moment dans le projet ou nous avions du mal à tester nos fonctions et l'interface de notre main ne nous convenais pas. Nous avons alors décidé de créer un espace de test de fonction qui serait plus facile et plus rapide pour tester différentes fonctions avec différentes entrées. Il nous est alors venu à l'esprit de faire un debug_mode qui nous permettrait de tester toutes nos fonctions facilement et rapidement.

3.1 Structure du programme

Notre debug_mode se présente donc comme un répertoire de l'ensemble de nos fonctions. Il permet d'y faire appel via une interface prenant en charge le choix des arguments (listes, entiers, chaines de caractères). Il prend également en compte une gestion des fichiers de sauvegardes. Les fonctions sont présentées dans un menu à choix multiples et rangées par rubriques. Par exemple,

les fonctions d'affichage sont rangées dans le premier choix du menu : "1.Fonctions d'affichage". Un sous-menu s'affiche alors pour sélectionner la fonction dans sa rubrique. L'utilisateur peut à tout moment revenir au menu principal avec le choix "r.Retour" et quitter le programme avec le choix présent dans le menu principal "q.Fermer debug_mode". Des variables sont également incluses dans le programme. Elles permettent de sauvegarder pendant la durée de l'utilisation du programme des cartes, des listes de cartes, etc ... Nous avons implémenté dans notre programme :

— 4 listes : "liCartes", "liCartes2", "liCartes3 et "liFichier"

— 2 dictionnaires : "carte1" et "carte2"

— une chaîne de caractères : "car"

— un entier (négligeable) : "entier"

Les listes servent de listes de cartes, c'est à dire de listes de dictionnaires, sauf "liFichier" dont la fonction sera détaillée plus tard (cf. **??**). Les dictionnaires représentent des cartes conforme à celles que le sujet demande.

Ces variables peuvent être utilisées à la guise de l'utilisateur via les différentes fonctions. Le menu principal est accompagné d'un tableau affichant les valeurs des différentes variables. Pour éclaircir notre code, nous avons compartimenté l'affichage de ce tableau dans la fonction "*tabVar*". Elle ne prend par conséquent aucun argument ni ne renvoie rien. Cette fonction fait d'abord appel à la fonction "*texte_encadre*" vue précédemment (cf.**??**) qui permet d'entourer un texte avec des '=' pour le mettre en valeur. Nous affichons donc le titre du tableau puis les différentes variables avec leurs valeurs. Comme le tableau s'affiche en même temps que le menu, si une variable viens à être modifié par une fonction, sa valeur mise à jour sera affichée dans le tableau à la fin de l'appel de la fonction.

Lorsqu'une fonction est sélectionnée dans le menu, les variables doivent être sélectionnées avant que l'appel à la fonction ne soit fait. La fonction "*choixVar*" permet d'afficher à l'utilisateur les variables pour qu'il puisse en choisir une. Elle prend comme argument une chaine de caractère représentant le type de variable que l'utilisateur doit choisir. Ainsi, si l'utilisateur veut réaliser un appel à la fonction "*carte_to_chaine*", prenant pour unique argument un dictionnaire, le programme principal de debug_mode fera appel à sa fonction "*choixVar*" avec en argument la chaîne de caractère "c" (pour carte). Un menu s'ouvrira pour que l'utilisateur puisse choisir entre "carte1" et "carte2". Un appel à la fonction "*carte_to_chaîne*" sera ensuite réalisé dans le programme principal avec pour argument l'un des deux dictionnaires choisi.

3.2 Gestion des fichiers

Certaines fonctions peuvent cependant prendre comme argument des noms de fichiers, nous avons donc mis en place une gestion dynamique des fichiers présents dans le sous-répertoire "ressources" (cf.**??**). La gestion des fichiers de notre programme s'articule autour de la fonction "*choixFichier*". Cette dernière permet à l'utilisateur de choisir un fichier existant présent dans le dossier "ressources", ou bien d'en créer un nouveau si c'est nécessaire. Afin que la gestion des fichiers soit dynamique et que par exemple, un fichier crée avec cette fonction puisse par la suite être sélectionné par cette même fonction, nous avons créé une liste "liFichier" qui répertorie l'ensemble des fichiers consultables. La fonction "*choixFichier*" consiste donc à lire ou modifier la liste "liFichier" ou fonction de ce que l'utilisateur choisi. Elle prend donc pour argument la liste à modifier (liFichier), retourne le nom du fichier choisi ou crée et a comme effet de bord la modification de la variable "liFichier".

Notre fonction "*choixFichier*" fait appel à plusieurs autres fonctions afin de compartimenter des actions spécifiques. La fonction "*affich_liFichier*" permet l'affichage de la liste "liFichier" sous la forme d'un menu numéroté. En effet, cette fonction a été réalisé dans l'optique d'être appelé par "*choixFichier*" juste avant un input de l'utilisateur pour choisir un fichier existant. Elle n'a donc pas d'arguments ni de valeur de sortie et a pour effet de bord l'affichage.

L'autre fonction appelée par "*choixFichier*" permet la sauvegarde sur le long-terme de la modification de la liste "liFichier". Cette dernière possède une sauvegarde de ses valeurs dans le fichier "liFich.csv". Ainsi, lorsque "liFichier" est modifiée par "*choixFichier*", la nouvelle version de "liFichier" est écrite dans "liFich.csv". Afin que cette sauvegarde soit persistante lors des redémarrages du debug_mode, la fonction "*init_liFichier*" est appelé lors de l'initialisation de la variable "liFichier" par le programme principal. Elle ne possède pas d'arguments, ne réalise pas d'effets de bords et renvoie une liste correspondante à "liFichier". Elle permet la lecture du fichier "liFich.csv" pour collecter tout les nom de fichiers présents dans le dossier "ressources" et les envoyer sous forme d'une liste.

3.3 Limites et Erreurs

Lors du développement de debug_mode, nous avons été confrontés à de multiples problèmes. Le premier, et selon nous plus important a été le manque de préparation et d'organisation dans la réalisation de la structure du programme. Nous n'avons pour ainsi dire pas réfléchi à l'architecture du programme, ou comment les différentes fonctionnalités de ce programme allaient exister entre elles. Alors que nous avions bien en tête l'objectif principal du programme, c'est-à-dire une interface utilisateur permettant de faciliter l'appel de fonctions pour les tester, nous n'avions pas encore réfléchi à comment réaliser cet objectif. Nous avons résolu ce problème au fur et à mesure du développement du programme. En résulte un programme manquant, à notre goût, d'une certaine rigueur dans l'exécution des tâches qu'il lui est demandé.

3.3.1 La redondance

La redondance dans les instructions est selon nous un symptôme de cette mauvaise préparation. Par exemple, lorsqu'une fonction retourne une valeur, le programme principal répète la même instruction pour tous les choix de variables de l'utilisateur :

Exemple 1. Sélectionnons une partie du code principal de debug_mode :

```
1  elif choix=="3":
2      chVar=choixVar("lc")
3      print("Création d'un jeu à 32 ou 52 cartes ?")
4      chNbCartes=int(input("(ecrivez 32 pour le jeu à 32, n'importe quelle autre
       nombre pour le jeu à 52)"))
5      if chVar=="lc1":
6              liCartes=fonctions.init_pioche_alea(chNbCartes)
7      elif chVar=="lc2":
8              liCartes2=fonctions.init_pioche_alea(chNbCartes)
9      elif chVar=="lc3":
10         liCartes3=fonctions.init_pioche_alea(chNbCartes)
11     else:
12         print("Erreur: variable inconnue, code retour choixVar() inconnu")
13     input("(Appuyer sur entrer pour revenir au menu)")
14     conti=False
```

Le code de la ligne 227 à la ligne 240 représente une redondance car seul le nom de la variable recevant la valeur retour change.

Nous aurions pu pallier cette redondance en regroupant l'ensemble des variables pouvant être sélectionnées par l'utilisateur dans une liste ou un dictionnaire. Cela nous aurait permis d'éviter la redondance de nos instructions car le choix de l'utilisateur aurait pu se porter sur l'indice d'une liste ou la clé d'une entrée de dictionnaire.

Exemple 2. Reprenons le dernier exemple.

```
1  nomvar=[
2      "carte1",
3      "carte2",
4      "liCartes",
5      "liCartes2",
```

```
6       "liCartes3",
7       "car"
8   ]
9   var=[
10      {"valeur":"A","couleur":"C"},
11      {"valeur":"R","couleur":"T"},
12      [{"valeur":9,"couleur":"C"},{"valeur":10, "couleur":"K"},{"valeur":9,"couleur":
        "T"}],
13      [],
14      [],
15      "random"
16  ]
17
18  chVar=int(input("Quelle variable? (0,1,2,3,4 ou 5):"))
19  print("Création d'un jeu à 32 ou 52 cartes ?")
20  chNbCartes=int(input("(ecrivez 32 pour le jeu à 32, n'importe quelle autre nombre
        pour le jeu à 52)"))
21  var[chVar]=init_pioche_alea(chNbCartes)
22  input("(Appuyer sur entrer pour revenir au menu)")
23
24  print(var[chVar])
```

Nous avons ici utilisé une paire de liste, "nomvar" pour le nom des variables et "var" pour la valeur des variables. Comme vous pouvez le constater, nous n'avons plus besoin de nous répéter dans les instructions. Cette méthode meilleure en tout point à celle mise en place dans la version actuelle du programme aurait néanmoins nécessité une réécriture complète du code de debug_mode pour être effective. Pour des raisons de gestion du temps, nous n'avons pas pu nous le permettre.

3.3.2 Une gestion partielle des fichiers

Un autre symptôme du manque d'architecture dans la réalisation d'un programme est l'ajout successif de fonctionnalités non prévues. Un gestionnaire de fichiers s'est imposé lorsque nous avons essayé d'implémenter les fonctions "*init_pioche_fichier*" et "*ecrire_fichier_reussite*" dans notre interface utilisateur. Il devait pouvoir sélectionner le fichier désiré pour pouvoir le lire avec "*init_pioche_fichier*" mais également en créer de nouveaux pour pouvoir sauvegarder des listes de cartes avec "*ecrire_fichier_reussite*". Nous avons donc dans un premier temps créé "liFichier" permettant de pouvoir garder en mémoire le nom des fichiers créés par l'utilisateur. Cependant, cette mémoire était à court-terme car lorsque le programme se fermait, les modifications apportées à "liFichier" n'étaient pas sauvegardées. Nous avons donc dans un second temps créé un système de sauvegarde de la liste dans le fichier "liFich.csv". Malgré cet ajout, la gestion des fichiers reste partielle : l'utilisateur ne peut par exemple pas supprimer de fichiers via le debug_mode. Ces défauts peuvent néanmoins se justifier par le fait que debug_mode ne s'adresse pas à des utilisateurs "grand-public" mais plutôt à des utilisateurs connaissant et ayant accès au code de l'ensemble du projet (des développeurs).

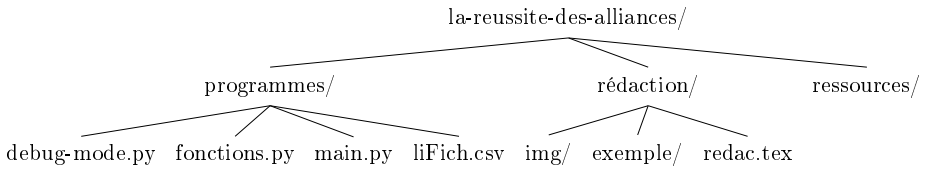
3.3.3 Une syntaxe conditionnelle lourde.

Enfin, nous avons également pu relever que la syntaxe "if/elif/else" pouvait paraître elle aussi redondante, ou tout du moins lourde. Nous avons cherché des solutions en utilisant d'autres syntaxes conditionnelles comme "match-case". Semblable à la syntaxe "switch-case" en langage C, "match-case" nous aurait permis une syntaxe plus légère pour l'écriture de nos menus et sous-menus. Malheureusement, cette fonctionnalité est sortie avec le patch 3.10 de Python en octobre dernier. Jugeant cette date beaucoup trop proche et afin d'éviter d'éventuels conflits de versions de Python, nous avons renoncer à implémenter cette fonctionnalité.

4 Annexes

4.1 Structure du répertoire

Cette arbre montre la structure de notre répertoire.



4.2 Fonction "afficher_reussite_num" au commit 728144c

```
1  def afficher_reussite_num(liCartes):
2      i=0
3      li_saut=[]
4      for carte in liCartes:
5          print(carte_to_chaine(carte),end=" ")
6          if carte["valeur"]==10:
7              li_saut+=[True]
8          else:
9              li_saut+=[False]
10     print()
11     for saut in li_saut:
12         if saut:
13             print("^"*3,sep="",end=" ")
14         else:
15             print(" ","^"*2,sep="",end=" ")
16     print()
17     cpt=1
18     for saut in li_saut:
19         if cpt<10:
20             print(" "*2,cpt,sep="",end=" ")
21         else:
22             print(" ",cpt,sep="",end=" ")
23         cpt+=1
24     print("\n")
```

fonction "afficher_reussite_num" au commit 7281444cd0ad4bd01bbffb3b81c877c213a2aeac

4.3 Sources

— xm1math.net

— stackoverflow.com

— overleaf.com