

INF101 : Projet - Jeu de Blackjack

Dans ce projet on se propose de réaliser un jeu de cartes : le Blackjack. Certaines fonctions sont guidées, mais vous pouvez (et devez) aussi laisser libre cours à votre imagination pour ajouter des fonctionnalités, une interface, etc. Après la fin de la section B, il n'est pas nécessaire de suivre exactement l'ordre des questions (par exemple il est possible de coder une interface graphique avant d'avoir intégré des joueurs intelligents).

Ce projet se réalise en binôme. Vous devrez rendre votre code et rapport avant le dernier TP (semaine du 10 décembre), lors duquel aura lieu une soutenance / démonstration de votre projet.

1 Introduction

1.1 Règles du jeu

On s'intéresse à une version simplifiée du jeu de Blackjack, qui se joue avec plusieurs paquets de 52 cartes. Chaque paquet de 52 cartes est divisé en 4 couleurs (pique, coeur, trèfle, carreau), avec 13 cartes par couleur, numérotées de 1 (as) à 10, puis valet, dame, roi.

Le but d'une partie de Blackjack est de s'approcher le plus possible de 21 points, sans dépasser ce total. Les valeurs numériques des cartes sont les suivantes :

- Les cartes numérotées de 2 à 10 valent autant de points que leur numéro (donc entre 2 et 10 points)
- L'as vaut soit 1 soit 11 points, au choix du joueur
- Les figures (valet, dame, roi) valent chacune 10 points

Les joueurs commencent la partie avec 2 cartes. A son tour chaque joueur peut choisir de s'arrêter avec son total actuel, ou de piocher une nouvelle carte (face cachée) pour l'ajouter à son total. Si le nouveau total dépasse alors 21 points, il perd immédiatement. Si le total égale 21 points exactement, il gagne immédiatement. Dans les autres cas, il reste en jeu et pourra à son prochain tour à nouveau choisir de continuer ou s'arrêter. Quand plus aucun joueur n'est en jeu (tous ont arrêté ou perdu) alors le gagnant est le joueur le plus proche de 21. Remarque : une version plus complète des règles du Blackjack se trouve ici : <https://www.guide-blackjack.com/Regles-du-black-jack.html>

1.2 Structures de données

On veut pouvoir jouer plusieurs parties à la suite. On va donc manipuler plusieurs scores pour chaque joueur, qui devront donc être stockés dans des listes ou des dictionnaires différents.

- Le total de points de la partie en cours, remis à 0 au début de chaque partie, et qui vaut la somme des valeurs des cartes tirées par le joueur pendant cette partie. Si ce total dépasse 21, le joueur perd la partie.
- Le total de victoires ou d'argent gagné par le joueur après toutes les parties jouées. Dans un premier temps, on comptera 1 point par partie gagnée et 0 par partie perdue. Dans un deuxième temps les joueurs disposeront d'une somme d'argent initiale qu'ils pourront miser, et leurs gains (ou pertes) monétaires pour la partie seront proportionnels à leur mise.

Option 1 : on enregistre les scores, les victoires, les gains de chaque joueur dans des listes, et on accède donc au score d'un joueur à partir de son numéro (l'indice de son score dans la liste). Option 2 : on enregistre les scores, victoires, gains des joueurs dans un dictionnaire, dont les clés sont les noms des joueurs, et on peut donc accéder à un score à partir du nom d'un joueur plutôt que de son numéro.

2 Partie A : initialisations

2.1 A1 - paquet de cartes

1. écrire une fonction `paquet()` qui initialise et renvoie un paquet de 52 cartes. Cette fonction n'a aucun argument. Elle renvoie une liste de chaînes de caractères contenant les noms des 52 cartes, par exemple "as de carreau", "roi de pique", "2 de trèfle", "7 de coeur".

2. écrire une fonction `valeurCarte(carte)` qui reçoit un nom de carte (une chaîne de caractères au format précédent) et renvoie sa valeur numérique selon les règles ci-dessus. **Attention** au cas particulier de l'as, il faut interagir avec le joueur pour lui demander de choisir la valeur voulue (seulement 1 ou 11, attention à bien filtrer la saisie). *Indice* : la couleur de la carte n'étant pas importante, il faudra commencer par extraire le premier mot de la chaîne pour obtenir le nom.
3. écrire une fonction `initPioche(n)` qui reçoit en argument l'entier `n` qui représente le nombre de joueurs de la partie, et renvoie la pioche constituée de `n` paquets de 52 cartes. Cette fonction doit utiliser la fonction `paquet` ci-dessus pour générer chaque paquet, puis les mélanger dans un ordre aléatoire, et renvoyer la liste représentant la pioche. Rappel: utiliser le module `random`.
4. écrire une fonction `piocheCarte` qui reçoit une liste `p` de cartes (la pioche) et un argument optionnel `x` (le nombre de cartes à piocher, par défaut 1). Cette fonction pioche `x` cartes au sommet de la pioche `p` et renvoie la liste des cartes piochées. **Attention** : cette fonction doit donc modifier la liste `p` reçue en argument.

2.2 A2 - joueurs et scores

1. écrire une fonction `initJoueurs(n)` qui reçoit le nombre de joueurs `n`, demande à l'utilisateur le nom de chaque joueur, les stocke dans une liste de chaînes de caractères, et la renvoie.
2. écrire une fonction `initScores(joueurs,v)` qui reçoit la liste des noms des joueurs et une valeur initiale `v` (optionnelle, par défaut 0); construit un dictionnaire dont les clés sont les noms des joueurs et les valeurs leurs scores initialisés à la valeur `v`; et renvoie ce dictionnaire.
3. écrire une fonction `premierTour(joueurs)` qui reçoit la liste des joueurs, construit un dictionnaire de scores initialement nuls (en appelant la fonction `initScores`), pioche pour chaque joueur 2 cartes (avec la fonction `piocheCarte` ci-dessus), et ajoute leurs valeurs à son score. Cette fonction doit renvoyer le dictionnaire de scores ainsi construit et initialisé.
4. écrire une fonction `gagnant(scores)` qui reçoit un dictionnaire de scores et renvoie le nom du gagnant et son score. *Rappel* : le gagnant est le joueur dont le score est le plus proche de 21, tout en restant inférieur ou égal à 21.

3 Partie B : gestion de la partie

3.1 B1 - Tour d'un joueur

- Écrire une fonction booléenne `continue()` qui interagit avec l'utilisateur pour savoir s'il veut continuer ou arrêter. Cette fonction doit lire sa réponse, la filtrer pour qu'elle prenne seulement une valeur autorisée (de votre choix : oui/non, go/stop, etc). Cette fonction renvoie ensuite `True` si le joueur souhaite continuer, `False` s'il souhaite arrêter.
- Écrire une fonction `tourJoueur(j)` qui reçoit le nom d'un joueur, et qui gère un tour de jeu de ce joueur (attention à bien passer en arguments toutes les variables nécessaires; attention à bien gérer les effets de bord) :
 - Affiche: numéro du tour, nom du joueur, son total de la partie courante (inférieur à 21)
 - Lui propose de continuer ou d'arrêter et lit sa réponse
 - Si le joueur continue, pioche la première carte de la pioche, calcule sa valeur, l'ajoute au total, vérifie la victoire ou la défaite (égale ou dépasse 21)
 - Si le joueur a choisi d'arrêter ou a perdu, il faut le retirer de la liste des joueurs encore en jeu

3.2 B2 - Une partie complète

- écrire une fonction `tourComple` qui donne un tour de jeu à chacun des joueurs encore en jeu dans la partie courante. Il faut donc appeler la fonction précédente pour chacun. Attention, cette fonction modifie éventuellement la liste des joueurs, puisque les joueurs qui perdent ou abandonnent doivent en être retirés.
- écrire une fonction booléenne `partieFinie` qui détermine si la partie courante est terminée (renvoie `True`) ou pas (renvoie `False`)

- écrire une fonction `partieComplete` qui appelle en boucle la fonction `tourCompleet` jusqu'à ce que la partie soit terminée et le gagnant connu. Cette fonction doit alors mettre à jour les nombres de victoires des joueurs (dictionnaire `victoires`). On considère pour l'instant que le gagnant remporte 1 point pour sa victoire, et les autres joueurs 0.

3.3 B3 - programme principal

Écrire un programme principal qui :

- demande à l'utilisateur le nombre de joueurs
- initialise les variables nécessaires (paquet de cartes, liste de joueurs, dictionnaires...)
- fait jouer une partie complète
- propose alors de faire une nouvelle partie. Si oui: boucle en reprenant la listes de joueurs initiale; si non: se termine en affichant un message de fin.

3.4 B4 - les mises

On veut maintenant permettre aux joueurs de choisir en début de partie combien ils veulent miser, après avoir vu leurs 2 premières cartes. Le programme leur demande combien ils misent. Leur mise doit être filtrée pour être strictement positive, et inférieure à leur total d'argent. A la fin de la partie, le gagnant ramasse toutes les mises des autres joueurs et les ajoute à son total d'argent.

Au début du programme, chaque joueur démarre non plus avec 0 points, mais avec 100 kopecs qu'il pourra miser. Les joueurs qui n'ont plus de kopecs en stock sont éliminés pour les parties suivantes. Quand il n'y a plus qu'un joueur engagé, le programme ne propose plus de rejouer, et se termine en affichant le nom du vainqueur.

- Écrire une nouvelle version de votre programme pour permettre de joueur avec des mises (attention à bien conserver la version sans mises).
- Option: proposer aux joueurs de quitter la table entre les parties (et donc de partir avec les kopecs gagnés jusque là).

4 Partie C : un peu d'intelligence

4.1 C1 - le croupier (piocher ou ne pas piocher, telle est la question)

On rajoute maintenant le rôle du croupier, qui n'est pas joué par un joueur humain, mais par le programme. C'est donc une "intelligence artificielle". Dans cette première version on considère que le croupier mise toujours la même valeur (par exemple 10 kopecs), mais on veut écrire plusieurs fonctions représentant plusieurs manières de décider de continuer à piocher ou pas.

- Écrire une première version "aléatoire" du croupier, qui décide au hasard de continuer ou arrêter
- En faire ensuite une version paramétrée pour choisir un croupier plus ou moins "joueur" en modifiant la probabilité de continuer. Avec une probabilité à 1, le croupier prend le maximum de risques et continue toujours à piocher (il perd donc la plupart du temps en dépassant 21) ; avec une probabilité à 0, il ne prend aucun risque et arrête tout de suite (même avec 0 points, ce qui n'a donc aucun intérêt) ; avec une probabilité à 0.5, on retombe sur la version aléatoire précédente.
- Écrire une version un peu plus intelligente : sa probabilité d'arrêter est maintenant décidée selon l'écart entre son total actuel et 21. Plus il est proche de 21, plus il a de risque de dépasser, et donc plus il doit avoir de probabilité de choisir d'arrêter.
- Enfin écrire une (ou plusieurs) autre(s) version(s) qui utilise(nt) la stratégie de votre choix pour optimiser le comportement du croupier. On pourra par exemple tenir compte du total des autres joueurs, des cartes déjà piochées / restantes dans le paquet, etc.

On écrira une fonction pour chaque stratégie, en leur donnant des noms parlants (par exemple : stupide, aleatoire, risquetout, etc). Testez vos différentes stratégies en faisant jouer plusieurs parties à votre croupier intelligent.

4.2 C2 - le croupier (choix des mises)

Écrire différentes fonctions permettant au croupier intelligent d'utiliser différentes stratégies pour choisir sa mise initiale (sur base de ses 2 premières cartes). Le choix pourra tenir compte par exemple :

- de l'argent restant (sur son total initial de 100 kopecs qui a pu évoluer au fil des parties précédentes) : par exemple miser un pourcentage donné de l'argent disponible
- d'une valeur d'aversion au risque (toujours miser le moins possible, ou le plus possible, ou une valeur intermédiaire)
- du total initial des 2 cartes reçues (selon la probabilité de s'approcher de 21)
- des cartes reçues par les autres joueurs (qui sont visibles à tous)
- ou d'une combinaison de tous ces facteurs, voire d'autres que vous pourrez imaginer

4.3 C3 - les autres joueurs

Comme pour le croupier, ajouter la possibilité en début de partie de choisir pour chaque joueur s'il est joué par un humain ou par l'ordinateur. On veut aussi pouvoir choisir (pour ceux qui sont joués par l'ordinateur) le type de joueur (c'est-à-dire laquelle des stratégies ci-dessus il utilise) : aléatoire total, risque-tout, risque mesuré, etc. Selon la stratégie choisie par l'utilisateur pour ce joueur, la fonction appelée doit choisir la mise de départ, et prendre la décision de continuer ou arrêter à chaque tour. Normalement, on doit pouvoir appeler les fonctions définies en C1 et modifiées en C2.

4.4 C4 - tournoi automatique et comparaison des stratégies

Faire jouer plusieurs parties complètement automatiques à des joueurs utilisant vos différentes stratégies, pour comparer leur efficacité. Pour lisser l'aléatoire de la pioche, il faut faire jouer suffisamment de parties. On comparera ensuite les scores moyens de chaque stratégie.

BONUS : en utilisant matplotlib, on pourra dessiner une courbe de score moyen en fonction de la probabilité de rejouer pour le joueur "aléatoire probabiliste". Par exemple le score moyen avec une probabilité de rejouer de 0 est de 0 (ne marque jamais aucun point), de même que pour une probabilité de rejouer de 1 (ne s'arrête jamais à temps).

5 Partie D : interface graphique

Pour l'instant, la partie se joue depuis la console textuelle. Vous pouvez en faire une version graphique avec affichage des cartes, interaction en cliquant sur la pioche, etc.