Rapport jeu de cartes

MOREAU Quentin

28/05/2020

Sommaire

1 Présentation

1.1 Introduction

L'aléatoire est au coeur de la vie quotidienne. En revanche, en informatique, il n'y a pas de place pour le hasard. La simulation d'un comportement aléatoire fait donc l'objet de nombreuses recherches, qu'il s'agisse de génération procédurale ou de processus stochastiques. Ici, nous prenons l'exemple d'un jeu de cartes, à savoir la *Bataille*, pour présenter une méthode simple et efficace : **la méthode de Fibonacci.** Le projet est réalisé avec **le langage de programmation fonctionnelle OCaml** afin de mettre en valeur le contexte mathématique.

1.2 Règles du jeu

Afin d'éviter toute ambigüité, nous fixons les règles du jeu, ces dernières pouvant faire l'objet de nombreuses variantes. Un paquet de 52 cartes est mélangé puis l'on distribue équitablement les cartes faces cachées aux deux joueurs qui forment chacun un paquet. A chaque tour, les adversaires dévoile la carte placée au-dessus dudit paquet. C'est ce qu'on appelle une *guerre.* Le gagnant est determiné par la carte la plus forte, c'est-à-dire celle de *valeur* la plus élevée selon les normes usuelles, *l'as* étant le plus puissant. Si les deux cartes sont de même *valeur,* la *guerre* continue et les joueurs répètent l'opération précédente en ajoutant une carte par-dessus celles déjà placées. Les joueurs réalisent donc l'opération en boucle jusqu'à ce que l'un d'eux sorte victorieux ou qu'il n'ait plus de carte. Le gagnant d'une *guerre* récupère les cartes du haut vers le bas de la pile puis les place en-dessous de son paquet. A chaque tour, les joueurs entamment un nouvelle *guerre* selon les mêmes consignes. Au bout d'un nombre de tours définis au préalable, le jeu prend fin. Si un joueur vide son paquet au cours de la partie, le jeu prend fin également. Le gagnant est alors celui possédant le plus de cartes. En cas d'égalité, on déclare match nul.

2 Représentation des données

Nous représentons les données principale sous forme de structures, auxquelles nous assignons des fonctions.

2.1 Structures

Dans un jeu de 52 cartes, chaque carte peut être identifiée de manière unique par 2 caractéristiques, sa *valeur* et sa *famille*. On utilise donc les **types sommes** afin de **filtrer** les différents cas possibles. Une fois les types *rank* et *suit* créés, on définit simplement le type *playingCard* comme un tuple composé de ces deux types.

Remarque : Dans l'exemple de la Bataille, on pourrait se passer de la famille qui n'intervient pas dans les règles, mais on la représente pour d'éventuelles modifications.

On chercher ensuite à modéliser les *paquets de cartes* ainsi que la *guerre* ayant lieu à chaque tour. Lorsque le joueur retire une carte de son paquet, il retire systématiquement celle du haut. Lorsqu'il ajoute une carte, à l'inverse, il la pose sous son paquet. Ce procédé obéit à une structure **FIFO.** On choisit donc de représenter les paquets des joueurs comme des **files** de cartes. Quant à la guerre, les joueurs placent les cartes les unes par-dessus les autres puis le gagnant récupère les cartes du haut de la pile vers le bas. Ce procédé-ci obéit à une structure **LIFO.** On choisit donc de représenter les guerres comme des **piles** de paires de cartes. Les implémentations en **OCaml** de ces structures sont identiques au nom près, les différences étant réalisées au niveau des fonctions. En effet, un **enregistrement** composé d'un entier modifiable stockant le nombre d'éléments et une liste modifiable stockant le contenu représentent aussi bien l'une ou l'autre des structures. On emploie ici des types **polymorphes** afin de mettre en évidence la généralité des structures utilisées.

2.2 Fonctions

2.2.1 Cartes

Dans le cadre du jeu, les fonctions associées au type *playingCard* ont pour but de lier chaque carte à un *rang* entier. Un simple **filtrage** exploitant le type **somme** permet de renvoyer le *rang* d'une carte, c'est-à-dire sa *valeur* pour une carte de valeur numérique et respectivement 14, 13, 12 et 11 pour *l'as,* le *roi,* la *reine* et le *valet.*
On réalise également une fonction qui associe à chaque carte un unique entier de $[\![1, 52]\!]$, ainsi que sa fonction réciproque.

2.2.2 Piles et files

Les fonctions associées au **piles** et **files** sont similaires excepté **l'empilage** et **l'enfilage.** Dans les deux cas, on réalise une fonction renvoyant un booléen testant

si la structure est vide, ainsi qu'une fonction permettant de retirer la tête de la liste mutable tout en mettant à jour les informations. Pour ce qui est de l'ajout d'un élement, on réalise **l'adjonction** dans le cas de la **pile** et la **concaténation** dans le cas de la **file**.

3 Simulation de l'aléatoire

Afin de simuler un mélange de cartes, nous créons une séquence pseudo-aléatoire de nombres par **la méthode de Fibonacci**.

3.1 Méthode de Fibonacci

Commençons par décrire la méthode.
Il faut choisir une valeur initiale ou **graine** ainsi qu'une valeur maximale non incluse.
Etant donné une graine $\mathfrak{G}$ et une borne supérieure non incluse M On définit la pseudo-suite de Fibonacci $(\varphi_n)_{n\in\mathbb{N}}$ par :

$$\begin{cases} \varphi_0 = 1 \\ \varphi_1 = \mathfrak{G} \\ \forall n \in \mathbb{N},\ (\varphi_{n+2} = \varphi_{n+1} + \varphi_n)\ [M] \end{cases}.$$

Remarque : Ceci n'est qu'une variante parmi d'autres. On pourrait notamment choisir deux graines pour définir les deux premiers termes.

3.2 Algorithme de Fisher-Yates

L'algorithme de Fisher-Yates permet de réaliser une permutation au hasard. Nous nous en servons pour mélanger les cartes. On suppose la **méthode de Fibonacci** parfaitement aléatoire et on note $\varphi_M \in \mathbb{N}^{\mathbb{N}}$ la pseudo-suite de Fibonacci de borne supérieure M et de graine $\mathfrak{G}$ fixée. On suppose qu'il y a $n \in \mathbb{N}$ éléments à permuter, identifiés au nombres de l'ensemble $[\![0, n-1]\!]$. On construit la permutation $\sigma \in \mathfrak{S}_n$ par composition des transpositions :

$$\sigma = \prod_{k=0}^{n-1} \tau_{n-k,\varphi_{n-k}(k)}.$$

Remarque : Cet algorithme a une complexité temporelle linéaire, mais dans notre cas, on permutera toujours 52 cartes donc on considérera la complexité temporelle constante.

3.3 Implémentation

L'aléatoire a pour but de mélanger le jeu de cartes. On l'implémente de manière **impérative** pour tirer profit de la souplesse des **vecteurs.**

3.3.1 Application au jeu de cartes

Une fois les fonctions appliquant les deux algorithmes réalisées, il est facile d'en creér une nouvelle pour mélanger un paquet de cartes. On commence à créer un

vecteur de structures de type *playingCard*, composé des 52 cartes d'un jeu classique grâce aux fonctions associées au type en question. Ensuite, on permute les 52 cartes par **l'algorithme de Fisher-Yates** et de **Fibonacci.** On peut répéter l'opération plusieurs fois pour améliorer le mélange. Il suffit pour finir de convertir le **vecteur** en **pile**.

3.3.2 Complexité temporelle et spatiale

L'usage de **références** permet une **complexité spatiale constante.** De plus, le nombre d'éléments à permuter étant constant et le calcul des termes de la pseudo-suite de Fibonacci se faisant en parallèle, le seul paramètre variant est le nombre de mélanges. Si m est le nombre de mélanges, alors la **complexité temporelle** est de l'ordre de $O(m)$, donc elle est **linéaire.**

4 Conclusion

4.1 Simulations

Nous avons réalisé plusieurs simulations pour tester le programme. Elles sont répertoriées dans le tableau ci-dessous :

nombre de guerres	graine $\mathfrak{G}$	nombre de mélanges m	gagnant
20	62	10	joueur 2
100	62	10	joueur 1
100	61	10	joueur 2
500	61	10	match nul
500	61	3	joueur 2
10	61	3	joueur 1

4.2 Bilan

Les simulations aboutissent à des résultats différents donc on peut estimer le rôle du programme accompli. En revanche, il y a peu d'exigeances dans le cas de cette simulation mis à part l'apparence. Dans une situation plus stricte, il faudrait améliorer la méthode voir en adopter une nouvelle. On pense entre autres à la cryptologie où la stochasticité des générateurs aléatoire est très importante.

4.3 Possibilités d'améliorations

Dans le cadre du jeu de cartes, on peut étudier les différentes variantes de la **méthode de Fibonacci** afin de déterminer la plus judicieuse. On peut également utiliser l'heure de la machine afin de générer aléatoirement une graine et rendre les simulations autonomes. Enfin, on peut transposer ce programme pour un autre jeu et créer davantage d'interraction avec l'utilisateur.