Les 7 merveilleuses merveilles du merveilleux monde des 7 merveilleuses couleurs

Aurèle Barrière & Nathan Tomasset

8 mars 2016

Intoduction

Nous présentons dans ce projet une implémentation du jeu des 7 couleurs dont les règles seront décrites plus bas. L'enjeu est d'implémenter en C le jeu, puis de créer des stratégies. Nous proposerons une implémentation organisée, puis différentes stratégies pour jouer. Enfin, nous comparerons ces stratégies en les faisant s'affronter.

Règles du jeu

Le jeu se présente comme un tableau carré de 30 case de côté (ce nombre est paramétrable dans notre implémentatoin). Initialement, il est rempli aléaoirement avec 7 "couleurs" (ou lettres). Dans deux angles opposés, il y a une huitième et une neuvième couleurs : celles des joueurs. Tour à tour, les joueurs choisissent une couleur parmi les 7. Toutes les cases de cette couleur et adjacente à la couleur du joueur (ou adjacente à une case qui vient de changer de couleur à ce tour par cette méthode) prennent désormais la couleur du joueur. Le but est de posséder plus de la moitié du plateau.

Organisation du projet

Voir le monde en 7 couleurs

2.1

Pour initialiser le tableau de manière aléatoire, on parcourt la matrice (initialisée avec des 0), et on remplace chaque coefficient par une des 7 couleurs.

Pour obtenir ce nombre, on importe le module `stdlib`, on initialise le générateur pseudo-aléatoire avec le temps (`srand(time(NULL));`) et on prend le résultat de la fonction `rand()` modulo le nombre de couleurs.

Cependant, il faut bien initialiser l'aléatoire au début de la fonction principale, et pas à chaque partie. En effet, dans la suite on lancera plein de parties en même temps, qui seront donc identiques si lancées à moins d'une seconde d'intervalle.

Enfin,`rand()` modulo un nombre de couleurs ne produit pas un résultat avec une probabilité parfaitement uniforme, si le nombre d'entiers accessibles par `rand()` n'est pas un multiple du nombre de couleurs. En effet, soit d le nombre de couleurs. Si par exemple $k \times d - 1$ couleurs sont accessibles par `rand()`, on aura k entiers dont le résultat du modulo donne 0, mais seulment $k-1$ dont le résultat donne $d-1$. Cependant, vu le petit nombre de couleurs et le grand dombre d'entiers accessibles avec `rand()`, ce n'est pas un problème significatif.

2.2

On créée une fonction `void update_board(char player, char color, char * b)`. Elle prend en argument un joueur `player`, une couleur jouée `color` et un plateau `b`.

Elle consiste à parcourir la matrice, et si on trouve une case qui est de la couleur jouée et à côté d'une case de la couleur du joueur, on la remplace et on indique dans une variable qu'il y a eu un changement. S'il y a eu changement, il faut recommencer le parcours de la matrice.

On pose n la taille du tableau. Dans le pire cas, il faut réappliquer $2 \times n$ fois le parcours de matrice : à chaque parcours, il y a un changement. On a donc une complexité en $\mathcal{O}(n^3)$ dans le pire cas.

2.3

À la conquête du monde

3.1

On crée donc une fonction `player_choice`, qui demande à un joueur un caractère correspondant à une couleur. Si le caractère ne correspond pas à une de couleurs disponibles, on redemande jusqu'à obtenir un résultat convenable.

3.2

On implémente la fonction `int score (char * b, int color)` qui prend en argument un plateau de jeu et la couleur d'un joueur. elle se content de compter le nombre de cases de cette couleur pour calculer le score d'un joueur.

On calcule la limite de score à atteindre : si n est la taille du plateau, on prend $\frac{n^2}{2}$ (arrondi à la valeur supérieure).

Grâce à cette fonction, on peut donc vérifier à chaque tour si :

— un des joueurs a atteint la limite de score

— les deux joueurs ont le même score qui est égal à la limite (égalité)

Ainsi, on peut mettre des conditions d'arrêt au jeu. On peut également calculer le pourcentage en calculant $\frac{score \times 100}{n^2}$.

La stratégie de l'aléa

4.1

Dans les fichiers de stratégies (`strategy.c, strategy.h`), on implémente une stratégie aléatoire : de la même manière qu'on choisissait une couleur aléatoire pour initialiser le plateau, on retourne une couleur aléatoire que le joueur artificiel va jouer.

4.2

On implémente une fonction `char alea_useful_colors(int player)` qui va opérer ainsi :

On initialise un tableau de la taille du nombre de couleurs, initialisé avec des 0. Il contiendra des booléens qui spécifieront si une couleur est utile ou non.

On parcourt le plateau de jeu. Si on tombe sur une couleur à côté d'une case occupée par le joueur, dans le tableau précédent on indiquera que cette couleur est utile (c'est à dire que le joueur progressera s'il la joue).

On caclule le nombre de couleurs utiles. On tire un nombre aléatoire plus petit que ce nombre, et on renvoie la couleur correspondante.

La loi du plus fort

5.1

Pour implémenter la stratégie `greedy`, nous avons utilisé le deuxième plateau `test_board`. Pour chacune des couleurs, on va copier `board` dans `test_board` (avec la fonction `copy_board()`). Puis on va simuler un coup du joueur en utilisant la fonction `update_board` décrite plus tôt, sur le plateau `test_board`. Pour chacun de ces tests, on calcule le score obtenu et on l'écrit dans un tableau.

Enfin, on cherche l'indice maximum de ce tableau : c'est la couleur à jouer.

5.2

Pour qu'un affrontement soit équitable, il faut alterner le premier joueur. En effet, commencer donne un avantage (pour s'en convaincre, s'imaginer le jeu à 1 ou 2 couleurs ou encore lancer des compétitions utilisant la même stratégie). On va tirer aléatoirement le premier joueur. Avec un grand nombre de parties, on devrait tendre vers un résultat équitable.

Le côté est également important. On va donc générer seulement la moitié de la matrice, puis la répliquer. La matrice est donc symétrique suivant la diagonale qui ne contient pas les positions initiales des joueurs.

On aurait également pu faire pour chaque match le match retour, où on inverse le côté et le premier joueur.

5.3

On a fait s'affronter en 100 parties le joueur aléatoire et le joueur `greedy`.

Le joueur `greedy` les a toutes remportées.

Ensuite, on fait s'affronter en 100 parties le joueur `alea_useful` et le joueur `greedy`.

Le joueur `greedy` les a toutes remportées.

Les nombreuses huitièmes merveilles du monde

6.1

On implémente la fonction `frontier` qui, prenant en argument un joueur et un plateau, calcule le nombre de cases d'une couleur différente de celle du joueur mais à côté d'une case occupée par le joueur. Il s'agit de la frontière de la zone occupée par le joueur.

La stratégie consiste à maximiser cette valeur.

On implémente donc la fonction `hegemony` de la même manière que le glouton, mais en ne calculant pas le score mais bien la frontière.