Les 7 merveilleuses merveilles du merveilleux monde des 7 merveilleuses couleurs

Aurèle Barrière & Nathan Tomasset

8 mars 2016

Table des matières

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 intelligences artificielles basées sur différentes stratégies pour y jouer. Nous proposerons une implémentation organisée, puis différentes stratégies. Enfin, nous comparerons ces stratégies en les faisant s'affronter sur des grilles aléatoires.

Règles du jeu

Le jeu se présente comme un tableau carré de trente cases de côté (ce nombre est tout à fait arbitraire et paramétrable dans notre implémentation). Initialement, il est rempli aléatoirement avec sept "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 sept premières. 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

Le projet est découpé de la manière suivante :

Dans un fichier header `defines.h`, on déclare les constantes pour le jeu : taille du plateau, nombre de couleurs, couleurs des joueurs. Ce fichier sera à inclure dans tous les fichiers utilisant ces constantes. L'avantage est de pouvoir utiliser ces constantes dans tout fichier. Il faut cependant recompiler d'autres fichiers lorsqu'on change celui-ci. Ainsi, les variables globales qui sont susceptibles de changer seront déclarées globalement ailleurs.

Dans un fichier `board.c`, accompagné de son header `board.h`, on déclarera globalement les plateaux (le plateau réel, et un autre utilisé pour simuler des coups). On mettra également toutes les fonctions nécessaires à manipuler ces tableaux : `set, get` pour manipuler les valeurs, `update_board` pour mettre à jour un plateau après un coup, ainsi que les calculs de score ou de frontière par exemple.

Dans un fichier `strategy.c`, accompagné de son header `strategy.h`, on déclare toutes les fonctions qui prennent en argument un joueur, et retournent le coup à jouer en suivant une certaine stratégie : aléatoire, gloutonne, hégémonique...

Enfin, dans un fichier `7colors.c`, on dispose de la fonction de jeu, et une fonction principale qui l'appelle un certain nombre de fois.

Remarque : Nous avons choisi de ne pas redéfinir de types. En effet, nous aurions pu utiliser des structures pour les couleurs, les plateaux et les joueurs. Mais finalement, chacune d'entre elle ne demande rien de plus que le type de base utilisé : une couleur n'est rien de plus qu'un `char` (on affiche des lettres), un joueur n'est rien de plus qu'une autre couleur,

et un plateau n'est qu'une matrice de couleurs (un vecteur de vecteurs de `char`). Renommer ces types est finalement moins intuitif que les utiliser ainsi.

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 de nombreuses 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, au vu du faible nombre de couleurs et du nombre important d'entiers accessibles avec `rand()`, il ne s'agit pas ici d'un problème significatif.

2.2

On crée 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 n^2 fois le parcours de matrice : à chaque parcours, il y a un changement. On a donc une complexité en $\mathcal{O}(n^4)$ dans le pire cas.

Montrons que dans certains cas on doit réappliquer le parcours $\mathcal{O}(n^2)$ fois :

Dans le cas d'un serpent (voir FIGURE 1), si on applique l'algorithme depuis la case en bas à droite, chaque itération ne va colorier qu'une seule nouvelle case, puis se rappeler. Il va donc y avoir autant d'appels récursifs que de cases à colorier. Ce nombre de cases est un $\mathcal{O}(n^2)$: environ la moitié de la matrice.

Et on ne peut pas l'appliquer plus de n^2 fois : à chaque appel, on a changé la matrice donc on a ajouté au moins une case.

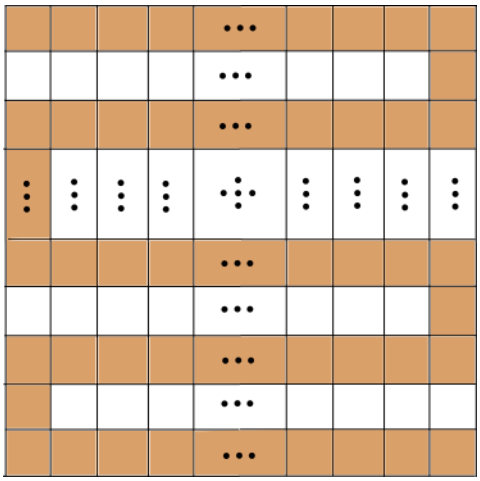

FIGURE 1 – Un exemple de cas pathologique pour notre algorithme

On en déduit que la complexité au pire de cette fonction est du $\mathcal{O}(n^4)$, où n est la taille du plateau de jeu.

2.3

On aurait pu implémenter une meilleure fonction de remplissage (voir Flood Fill Algorithms(lien externe)). Un des algorithmes consiste à récursivement regarder de chaque côté des cases atteintes si la case voisine est non coloriée et accessible.

On aurait ainsi pu réduire la complexité au pire à du $\mathcal{O}(n^2)$.

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 des couleurs disponibles, on redemande jusqu'à obtenir un résultat convenable.

Nous n'avons pas ajouté de couleurs ou d'interface graphique. Il aurait été possible d'ajouter de la couleur dans les fonctions `printf` de `print_board` avec les codes couleurs ANSI. Mais il aurait fallu définir la couleur de chaque caractère. Dans notre programme, si on veut ajouter des couleurs, il n'y a qu'à modifer le `#define NB_COLORS` et le programme utilisera tout seul les prochaines lettres de l'alphabet.

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 contente 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 inférieure : cela ne pose pas de problème, dans le cas d'un plateau avec un nombre de cases impaires, il ne peut pas y avoir d'égalité).

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 du plateau possédé par un joueur 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 avec des 0 un tableau de la taille du nombre de couleurs. 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.

Remarque : Il se peut que le nombre de couleurs utiles soit nul (si on a été complètement encerclé par l'adversaire). Il ne faut pas alors demander au programme de tirer un nombre aléatoire modulo 0.

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 où l'on commence est également important. On va donc générer seulement la moitié du plateau, puis le répliquer. Le plateau 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 cent parties le joueur aléatoire et le joueur `greedy`.

Le joueur `greedy` les a toutes remportées.

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

Le joueur `greedy` les a toutes remportées.

Plus de résultats seront donnés dans la section Tournois.

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.

Évaluation : Pour évaluer cette stratégie, on la fait jouer 1000 fois contre `greedy`.

`greedy` gagne 996 fois, `hegemony` gagne 4 fois.

6.2

On aurait également pu implémenter une stratégie de glouton prévoyant à une profondeur n. En effet, on aurait pu énumérer toutes les suites de n coups possibles, évaluer leur score

et prendre la première lettre de la séquence qui a le meilleur score.

Pour la gestion de la mémoire, on a un choix :

Soit on crée un tableau à chaque profondeur pour pouvoir revenir au coup précédent dans l'exploration des coups.

Soit on rejoue toute la séquence à chaque changement (moins d'espace, mais plus long puisqu'on rejouera souvent la même sous-séquence de coups).

Bonus : stratégie `starve`

Devant l'échec d'`hegemony` malgré ce qui nous semblait une bonne idée, à savoir chercher à s'étendre au maximum pour "affamer" l'adversaire en l'empêchant d'accéder à certaines cases qui sont alors garanties de nous revenir, nous avons cherché à mettre au point une nouvelle stratégie. L'idée est de chercher à maximiser non pas sa frontière ou son score mais l'"espace personnel", à savoir l'ensemble des cases inaccessibles par l'adversaire. Pour cela, on définit une fonction `available` qui calcule le nombre de cases accessibles pour un joueur. Il s'agit d'appliquer un algorithme de remplissage : à l'initialisation, seules les cases possédées sont accessibles, on re-parcourt ensuite tant que nécessaire l'ensemble des cases pour ajouter aux cases accessibles les cases non possédées par l'adversaire et dont un voisin est accessible, et ce jusqu'à stabilisation. On calcule ensuite l'espace personnel d'un joueur comme le nombre total de cases auquel on retire le nombre de cases accessibles par l'adversaire. La stratégie `starve` consiste enfin à maximiser cet espace personnel de la même manière que `greedy` maximise le score du joueur.

Stratégie générale

On constate que les implémentations de `starve` ou `hegemony` ressemblent fortement à celle de `greedy` : il suffit de changer la fonction à maximiser. Avec l'ordre supérieur, on peut avoir une fonction qui prend en argument une fonction à maximiser et renvoie la couleur à jouer pour maximiser cette fonction (`score` pour `greedy` et `frontier` pour `hegemony` par exemple). Il faut de plus que cette fonction ne choisisse que parmi les couleurs utiles (qui font avancer le score du joueur lorsque c'est possible), pour garantir la terminaison d'une partie.

On implémente donc la fonction `char maximize(int (*f) (char *, char), char player)`. Elle prend en argument une fonction à maximiser (`score`, `frontier` ou `personal_space`) et un joueur et renvoie la couleur à jouer.

Implémenter une stratégie se fait dès lors en une ligne :

```
char greedy(char player) { return maximize(score, player); }
```

Le pire du monde merveilleux des 7 couleurs

7.2

On implémente la stratégie hybride `greedymony`. Il s'agit pour elle de maximiser la somme du score et de la frontière. Elle bat toutes les stratégies implémentées jusqu'à présent.

En effet, le problème de `hegemony` est à la fin du jeu : toutes les zones entourées par notre couleur mais qui ne sont pas de notre couleur ne seront pas coloriées. Les colorier réduirait notre frontière, alors qu'on a intérêt à les prendre.

Le problème de `greedy`, lui, est de souvent favoriser la prise de cases qui ne lui permettent pas de gagner par rapport à son adversaire puisqu'elles lui étaient déjà inaccessibles.

La combinaison des deux est donc particulièrement efficace.

Si on fait un mélange entre `hegemony` et `starve`, on n'obtient pas une stratégie aussi forte : on accorde trop d'intérêt aux zones qui ne sont pas de notre couleur mais entourées par notre couleur puisqu'elles augmentent beaucoup la frontière et la zone personnelle, et ne seront donc jamais coloriées.

De même, un mélange entre `greedy` et `starve` n'est pas très intéressant, dans la mesure où le score actuel est déjà compris dans l'espace personnel : cela favoriserait fortement les couleurs qui augmentent le score en plus de l'espace personnel, or l'intérêt de `starve` est justement de faire perdre de l'intérêt à ces couleurs qui nous permettent de gagner des cases qui nous sont déjà garanties.

Tournois

Dans cette section, nous comparons les différentes stratégies en organisant des tournois de cent parties entre chaque paire de stratégies.Les parties sont jouées avec un terrain symétrique et un choix aléatoire du premier joueur pour garantir une équité maximale.

strategie VS	`alea`	`alea_useful`	`greedy`	`hegemony`	`starve`	`greedymony`	Total
`alea`	X	0	0	3	0	0	3
`alea_useful_colors`	100	X	0	49	0	0	149
`greedy`	100	100	X	100	31	2	331
`hegemony`	97	51	0	X	0	1	148
`starve`	100	100	68	100	X	4	368
`greedymony`	100	100	98	99	96	X	493

On constate que `greedymony` est la meilleure stratégie.

Ensuite, `starve` est la seconde meilleure stratégie (elle gagne contre `greedy` 68 à 31, et gagne 100 à 0 contre toutes les autres sauf `greedymony`).

Ensuite, `greedy` ne laisse gagner personne d'autre que `starve` et `greedymony`.

`hegemony` n'est pas très efficace, puisqu'elle est presque équivalente à `alea_useful_colors`.

Enfin, la stratégie `alea` n'est vraiment pas efficace et perd presque tout le temps.

Synthèse

Ce projet fut pour nous l'occasion de nous intéresser à plusieurs aspects.

Dans un premier temps, il fallait se fixer une architecture générale pour le projet. L'organisation en différents modules s'est révélée efficace.

Ensuite, le projet a soulevé des questions d'ordre algorithmiques (algorithmes de remplissage par exemple). Nous ne nous sommes que peu intéressé à la question de la complexité (seulement pour la mise à jour de plateau), cela ne nous semblait pas essentiel : chacune des stratégies peut être lancée une centaine de fois dans un temps raisonnable (moins d'une minute). La conception de stratégies fut une partie importante de notre projet.

Ce fut également l'occasion de s'intéresser à l'ordre supérieur en C.

Bibliographie

Flood Fill Algorithm :
`https://en.wikipedia.org/wiki/Flood_fill`

Higher Order in C :
`http://stackoverflow.com/questions/2535631/higher-order-functions-in-c`

Floating Point Exception :
`http://stackoverflow.com/questions/13664671/floating-point-exception-core-dump`

ANSI escape codes :
`http://stackoverflow.com/questions/3219393/stdlib-and-colored-output-in-c`