

Arcsys: le jeu des 7 couleurs

Rémi Hutin

Rémy Sun

28 février 2016

Résumé

Introduction

Question 2.1

La fonction `init_board` permet d'initialiser le monde des 7 couleurs (dans le module `tools.c` du code proposé).

Cette fonction parcourt le tableau `board`. A chaque case, on génère aléatoirement un nombre `n` entre 0 et 6 et on stocke dans la case `n + 'A'`.

La génération aléatoire de nombre se fait à l'aide la primitive `srand` de C et de la bibliothèque `time.h` utilisée pour initialiser le générateur de nombre.

Il s'agit ensuite d'initialiser les cases correspondant aux positions initiales des joueurs.

Avec cette façon de procéder, on efface génère aléatoirement un nombre pour les cases initiales, puis on l'écrase. Cela pourrait être évité en créant plusieurs boucles pour parcourir le monde, mais cela semble superflu.



*La fonction proposée dans le code ne fait pas exactement ce qui est décrit plus haut. Elle a été modifiée selon les réponses apportées à la **Question 5.2***

Question 2.2


La fonction `play` effectue ce qui est décrit dans l'algorithme de mise à jour du monde.

Le bon fonctionnement de la fonction a été effectué en testant son efficacité dans différents cas de figures.

Il est notamment possible de vérifier le bon fonctionnement de la fonction en sauvegardant l'état du monde avant et après utilisation de `play`. Il s'agit ensuite de contrôler que toutes les cases ne contenant pas la couleur jouée ont été inchangées, que toutes les cases à la nouvelle frontière du joueur courant ne sont pas de la couleur jouée et que l'ensemble des cases contrôlées par le joueur forment un ensemble connexe par arcs. Ce dernier point se vérifierait par exemple en vérifiant que chaque point contrôlé par le joueur est bien connecté à la position initiale (parcours récursif des voisins). Cependant une telle approche est très lourde, et il n'a pas paru judicieux d'inclure une vérification systématique à la fin de la fonction `play`.

Il faut noter que si une case est modifiée par l'algorithme à un parcours, il est possible que le prochain parcours change n'importe quelle case d'indice inférieur à cause de ce changement, les case d'indice supérieur auront déjà été parcourues lors du parcours courant.

Ainsi, le pire cas serait théoriquement 900 parcours : on mettrait à jour la dernière case de la dernière ligne, puis l'avant dernière case de la dernière ligne, et ainsi de suite. Une telle situation a cependant peu de chances d'arriver. On peut estimer que la pire situation réaliste est celle où, dans la configuration de départ du jeu, le premier joueur joue une couleur qui "serpente" sur le plateau, créant une situation où 199 tours sont nécessaires.

 La fonction proposée dans le code ne fait pas exactement ce qui est décrit plus haut. Elle a été modifiée selon les réponses apportées à la **Question 2.3**

Question 2.3

On crée une fonction propagateur qui fonctionne récursivement.

Quand on l'applique à une case, elle attribue cette case au joueur concerné, et pour chacune des cases voisines, si elles sont de la bonne couleur, elle s'invoque sur ces cases.

Il ne s'agit ensuite que de faire un seul tour de boucle en appelant cette fonction de propagation sur les cases de la bonne couleur voisines du joueur concerné.

Le bon fonctionnement de la fonction a été effectué en testant son efficacité dans différents cas de figures. Les mêmes règles de vérification de correction s'appliquent. On pourrait vérifier que le nombre de parcours par case est bien réduit en comptant le nombre où la fonction passe par chaque case.

Question 3.1


L'ensemble des fonction relatives à une partie entre humains est dans le module `tools.c`.

L'implémentation proposée ne peut cependant pas rattraper certaines erreurs humaines. Si un joueur s'obstine à jouer des caractères non valides, on ne peut pas programmer un moyen de lui faire jouer un coup valide.

Question 3.2

Une condition suffisante d'arrêt de la partie est l'occupation de plus de la moitié des cases par un joueur.


Les fonction afférentes à ce problèmes sont dans le module `tools.c`

 Il serait plus judicieux de compter dans les cases occupées celles entourées par un seul joueur. Une fonction a d'ailleurs été programmée pour repérer de telles cases dans le module `hegemon.c`. Néanmoins, cela nécessiterait de faire un appel à cette fonction à chaque tour, ce qui serait très lourd. De plus, si on comptait cela dans le pourcentage de territoires occupés, autant introduire une règle du type `go` qui convertit automatiquement les îlots de couleurs.

Question 4.1

Une fonction aléatoire a été programmée dans le module `random.c`. Elle utilise le même générateur de nombre que pour l'initialisation du monde.

Il est à noter qu'il est très probable avec cette méthode de ne rien jouer à chaque tour.

 La fonction proposée dans le code ne fait pas exactement ce qui est décrit plus haut. Elle a été modifiée selon les réponses apportées à la **Question 4.2**

Question 4.2

La fonction attendue a été programmée dans le module `random.c`

Question 5.1

La fonction attendue a été programmée dans le module `greedy.c`.

Elle procède de la manière suivante : pour chaque couleur, on joue la couleur puis on regarde le score obtenu. Ensuite on revient à l'état initial (préalablement sauvegardé).

On choisit la couleur ayant apporté le plus de points.

On peut évaluer la complexité de l'algorithme de la manière suivante :

- Une sauvegarde se fait en temps constant, comme une restauration. (90) Il y a 1 sauvegarde et 7 restaurations
- Dans chaque boucle on utilise `play`, qui passe au moins une fois par case, et qui fait appelle à `propagate`. On peut amortir la complexité des utilisations de `propagate` à 90×5 car la fonction effectue 5 actions (4 tests et une assignation) et à chaque fois que `propagate` est appelé sur une case, il ne peut plus être ré-appelé sur cette case (car c'est devenu une case joueur).
- il y a 7 boucles
- il faut faire au maximum 14 comparaisons + assignation pour garder fil de l'indice de la couleur maximale.

D'où une complexité en environ 3884 actions. De manière plus intéressante, si on compte n la taille du monde, la complexité est $O(n)$.

Question 5.2

Il y a deux facteurs principaux que nous avons identifiés rendant la situation des deux joueurs asymétrique (en excluant les différences de stratégies).

Le premier est le fait que la situation peut être très simple d'un côté (toutes les cases d'une même couleur sur un carré 4×4 par exemple), et très compliquée de l'autre (toutes les cases adjacentes sont de couleurs différentes par exemple). Il y a une solution simple pour pallier à ce problème : nous avons symétrisé le terrain.

Le second problème est plus épineux, et nous n'avons trouvé qu'une solution mitigeant un peu le problème sans le résoudre complètement. En effet, le joueur qui commence possède un avantage considérable (peu importe l'intelligence artificielle employée, le match-up miroir

est à +70% en faveur du joueur qui commence. La solution adoptée a été de rendre aléatoire le joueur commençant, afin d'équilibrer un peu les chances.

Remarque. *Pour les stratégies déterministe (glouton, ...) on serait en droit d'attendre un taux de victoire de 100% pour le premier joueur. Ce n'est pas le cas, probablement parce que le premier joueur est parfois amené franchir la "diagonale" du monde, rendant la situation asymétrique.*

Question 5.3

Sur un tournoi de 100 parties, le joueur glouton totalise le plus de victoires. En fait, il a généralement tendance à gagner 100 parties, de temps en temps, il'en n'en gagne que 99.

Pour pousser l'expérience, nous avons fait des tournois de 1000 parties. Il gagne toujours presque toutes les parties à l'exception d'une ou deux.

On notera que le (seul) avantage du joueur aléatoire est que puisqu'il joue aléatoirement, il peut éventuellement tomber sur la stratégie optimale. A l'évidence, cela n'arrive pas souvent.

Question 6.1

L'intelligence artificielle hégémonique est programmée en `hegemon.c`.

Pour créer cette stratégie, on a construit une fonction qui calcule récursivement si une case colorée est "contestée", c'est-à-dire si les deux joueurs peuvent l'obtenir. Nous implémentons ensuite une fonction qui calcule le nombre de case libres entourant la zone du joueur, en éliminant les cases non disputées.

Il s'agit ensuite de faire une comparaison entre les gains apportées par couleur si elle est jouée.

Dans le tournoi effectué avec le joueur glouton, le joueur glouton gagne plus de 95 fois sur 100.

En fait, l'algorithme hégémonique est très peu efficace car, s'il augmente très vite son territoire au début, il s'essoufle très vite. En effet, il vient un moment où il n'y a que très peu de mouvement qui augmentent le périmètre. De manière générale, prendre une case fait diminuer ou ne modifie pas le périmètre extérieur. Le principal cas où il le fait augmenter est celui où on crée une "ligne". Ainsi, passé un certain cap, l'algorithme évite de gagner de territoire, ce qui est contre-productif.

Remarque. *Une version du joueur hégémonique prenant en compte les territoires non contestés et comptant le périmètre en terme de "points de contact" et non de cases en contact semblait tenir un avantage sur le joueur glouton. Il est à noter que cette version utilisait glouton s'il n'était pas possible d'augmenter le périmètre...*

1 Question 6.2

L'intelligence artificielle prévoyante est programmée en `foreseeing.c`.


Il suffit de sauvegarder l'état initial du monde, puis, pour chaque couleur, jouer la couleur et ensuite le coup glouton et comparer les points obtenus pour chaque couleurs. La sauvegarde permet de revenir à l'état initial du tour.

Glouton était implémenté en complexité linéaire (en la taille du monde). La même analyse découle pour la complexité du joueur prévoyant, si ce n'est que glouton est encore exécuté dans la boucle mais la complexité resterait linéaire.

Par contre, si on considère m le nombre de couleurs comme étant aussi une variable, la complexité de glouton serait $O(mn)$ et celle de prévoyant $O(m^2n)$

L'intelligence artificielle prévoyant sur plus de deux tours a aussi été implémentée. C'est la stratégie nommée `oracle` dans le module `foreseeing.c`.

Il n'y a rien de notable à signaler, à part que la fonction est récursive : au lieu d'appeler glouton pour savoir quel coup jouer après le premier coup de test, elle s'appelle elle-même pour un tour de moins, joue le coup diagnostiqué, et ainsi de suite jusqu'à ne plus avoir à jouer que deux tours. A ce moment on appelle le joueur prévoyant, joue son coup, puis le coup du joueur glouton.

 Si le principe est simple, la complexité explose très vite ! en effet, si on considère p le nombre de coups à jouer en avance, la complexité est de l'ordre de $O(7^p)$!

Question 7.1

Il est possible pour chaque joueur de créer un pire cas.

Approche heuristique

C'est peut-être moins évident pour le joueur aléatoire puisque beaucoup de cas sont mauvais pour lui. Globalement, on peut considérer comme pire cas celui où, pour les premiers coups, tous sont équivalents (et mauvais). C'est-à-dire, le joueur est confronté au plus de couleurs différentes possibles.

Pour le joueur glouton, il suffit de créer une configuration qui va l'amener à "s'enfermer" dans les coins. Un exemple serait celui où les deux bords du coin où est le joueur suivent la configuration ABCDEFG avec des cases afférentes permettant d'enfermer le joueur dans les bords du monde.

Pour le joueur hégémonique, il a déjà naturellement tendance à créer des pointes, il n'est pas très difficile de créer une situation qui encourage cette situation.

Pour les joueurs prévoyants, il faudrait procéder de la même manière

Méthode algorithmique

L'approche proposée plus haut est peu satisfaisante.

On peut voir le problème ainsi : il y a un nombre fini de configurations initiales possibles (7^{465}) dans le monde symétrique envisagé. L'ensemble $\{s(E) \mid E \text{ configuration}\}$ est la fonction donnant le nombre de tours nécessaire au joueur pour contrôler plus de la moitié des cases est un ensemble d'entiers naturels fini

non vide. Il admet un maximum. On peut raisonnablement considérer que la configuration correspondante à ce pire score est le pire cas possible.

Cela prendrait un temps énorme, mais pour un joueur donnée, on pourrait simplement chercher le maximum de s...

Question 7.2

On propose l'approche suivante : on considère une combinaison linéaire du nombre de cases au périmètre et du nombre de territoires gagnés pour prendre la décision. Cette algorithme de décision se base sur plusieurs tours et on vérifie toujours si on ne peut pas juste terminer la partie avec glouton.