Rapport du projet 7 colors

Marco Freire et Clément Legrand
L3 Info ENS, 2017/2018

1 Introduction

Seven colors est un jeu d'ordinateur inventé par Dmitry Pashkov, développé en 1991. Le principe est simple: deux joueurs essayent de conquérir la plus grande surface d'un terrain de jeu constitué de cases juxtaposées, de sept couleurs différentes. Pour cela, chaque joueur choisit à son tour une couleur, et conquiert toute case de cette couleur adjacente à la zone qu'il possède. Nous nous sommes penchés sur l'implémentation de ce jeu, de ses mécanismes et de différents joueurs artificiels. Ce projet amène à comparer ces différents joueurs entre eux afin de rechercher une stratégie aussi bonne que possible. De plus, de part la taille du programme, il devient réellement nécessaire de modulariser le code et de le découper en plusieurs fichiers.

2 Réponses aux questions

2.1 Voir le monde en sept couleurs

Implémentation du type `board_d`

Il convient tout d'abord de créer un type `board_d` afin d'encapsuler un peu le code et de ne pas modifier le plateau de jeu n'importe comment. Ce type contient un tableau de caractères de taille `BOARD_SIZE`^{2}, ainsi que deux entiers: `num_cell_sup` et `num_cells_down` indiquant le nombre de cases possédées par le joueur `^` et le joueur `v` respectivement. Afin de maintenir cette structure, nous avons implémenté les méthodes `get_cell`, `get_num_cells_up`, `get_num_cells_down`, `set_cell`, ainsi que `board_create` et `board_free`, allouant un tableau de la bonne taille, et le libérant respectivement.

Question 1

La fonction `rand_board` prend en argument un pointeur vers un tableau, et initialise en place chacune des cases de celui-ci aléatoirement. Nous avons pour cela recours à la fonction `rand`. Celle-ci renvoie un `int` tiré uniformement (et donc compris entre 0 et $RAND\_MAX = 32767$), que nous ramenons entre 0 et 6 avec une division euclidienne. On notera que 32768 n'est pas divsible par 7, ainsi la lettre A est très légerement plus probable que les autres, mais ceci est négligeable (aux vues des dimensions du tableaux notamment).

Question 2

La fonction `update_board` prend en argument un pointeur vers le tableau de jeu, ainsi que le joueur courant et la couleur jouée. la fonction `print_board` affiche les valeurs des cases du tableau du pointeur qui lui est donné en argument. Elle permet de s'assurer sur quelques exemples que le tableau évolue bien de la manière souhaitée.

Complexité dans le pire cas : Dans le pire des cas, à chaque itération, l'algorithme met à jour une et une seule case. Posons $n =$ `BOARD_SIZE`. Étant donné qu'il y a n^2 cases dans le tableau, au plus n^2 passages sont effectués. Lors d'un passage, n^2 cases sont examinées, chacune d'elles en un temps constant. La complexité dans le pire cas de cet algorithme est donc un $O(n^4)$. Cette borne est réellement atteinte: par exemple si une couleur forme une spirale rectangulaire jusqu'au centre. La complexité est donc un $\Theta(n^4)$.

Complexité amortie : Le cas énoncé précédemment est assez pathologique et ne peut se produire trop souvent. L'analyse de la complexité amortie serait donc pertinente, bien qu'un peu délicate.

Question 3 (bonus)

Principe de l'algorithme implémenté : Plutôt que d'effectuer des parcours entiers du tableau en vérifiant si les cases ont besoin d'être mises à jour, il est préférable de se représenter la zone conquise par un joueur comme un graphe (non orienté), dont les sommets sont les toutes les cases du tableau et les arêtes lient chacune des cases conquises par le joueur aux quatre cases adjacentes sur le plateau. Il suffit alors d'effectuer un parcours de ce graphe en le mettant a jour progressivement.

Nous implémentons donc deux fonctions, la première `update_board_optimized` crée un second tableau `board_visited`, de mêmes dimensions, dont les cases indiquent si le sommet (i, j) a été visité ou pas. Nous parcourons ensuite récursivement le graphe en commençant par la case du coin (qui appartient nécessairement au joueur), à l'aide de la fonction `update_board_rec`. À chaque sommet visité, nous le marquons comme tel dans `board_visited`, et s'il est de la bonne couleur ou appartient au joueur courant, nous marquons la case comme appartenant au joueur dans `board` et explorons ses voisines non visitées.

Les fonctions `update_board_rec` et `update_board_optimized` opèrent toutes deux en place.

Complexité dans le pire des cas : Chaque case du tableau est visité un nombre fini borné de fois (au plus quatre dans notre implémentation), donc l'algorithme est un $O(n^2)$. Là encore, cette borne peut être atteinte si tout le tableau est de la même couleur. La complexité est donc un $\Theta(n^2)$.

2.2 À la conquête du monde

Question 4

Les fonctions nécessaires pour faire jouer un joueur contre un autre sont `player_input` qui permet de récupérer la couleur jouée par le joueur courant, et `change_player`. La fonction `game_loop` effectue alors l'essentiel du travail en faisant appel aux bonnes fonctions. Pour jouer à deux, il suffit de remplacer

```
case 'v':
    color_input = IA_random_wise(board, *player);
    *is_IA_turn = true;
    break;
```

par

```
case 'v':
    color_input = player_input();
    *is_IA_turn = true;
    break;
```

L'implémentation contraint les joueurs à s'affronter sur la même machine.

Question 5

La partie peut s'arrêter si l'un des joueurs au moins détient plus de la moitié du plateau de jeu. Nous avons donc recours aux attributs `num_cells_up` et `num_cells_down` du type `board_d`, afin de savoir si ceux ci excèdent la moitié du nombre total de cases. La fonction `is_end` vérifie cette condition, et la fonction `print_occupation` affiche le pourcentage du plateau occupé par chaque joueur.

2.3 La stratégie de l'aléa

Question 6

La fonction `IA_random` ne prend aucun argument et renvoie une couleur tirée au hasard.

Question 7

La fonction `IA_random_wise` fait de même en s'assurant que que la couleur jouée permet de gagner du terrain. On détermine pour cela quelles sont les couleurs admissibles, et nous tirons ensuite une couleur au hasard, jusqu'à que celle-ci soit admissible.

2.4 La loi du plus fort

Question 8

La fonction `IA_greedy` prend en argument un pointeur vers le plateau de jeu, ainsi que le joueur courant et renvoie la couleur choisie par l'algorithme glouton. Nous avons pour cela repris et légèrement modifié la fonction `update_board_optimized` impémentée pour la question 3: nous comptons désormais le nombre de cases modifiables en jouant une couleur donnée au lieu de les modifier. Nous effectuons ensuite le choix maximisant ce nombre de cases.

Question 9

Pour que le combat soit "équitable", le mieux est de faire s'affronter les deux joueurs artificiels sur un plateau donné, puis d'effectuer l'affrontement à nouveau, mais en échangeant les positions de départ.

Question 10

Sur les cent essais effectués (sur des grilles carées de coté 30, et en échangeant à chaque fois les rôles des joueurs), le joueur artificiel glouton à systématiquement gagné. Il pourrait être intéressant de déterminer expérimentalement l'influence de la taille de la grille.

2.5 Les nombreuses huitièmes merveilles du monde (bonus)

Question 11

Il apparaît facilement qu'en effectuant un parcours entier du plateau et en comptant les cases ayant parmi leurs voisines, une appartenant au joueur courant, il est possible d'obtenir le périmètre de la zone détenue par le joueur en $\Theta(n^2)$. Il suffit dès lors, pour chaque couleur possible, de créer une copie du plateau, d'effectuer sur celle- ci une mise à jour en jouant ladite couleur, et de compter le périmètre correspondant. La complexité de cet algorithme est donc un $\Theta(n^2)$. La fonction modélisant l'algorithme hégémonique est `IA_greedy_borde`
En confrontant cet algorithme avec l'algorithme glouton, il apparaît que ce dernier est meilleur (quasi intégralité des parties gagnées).

Question 12

La fonction modélisant le glouton prévoyant est `IA_foresighted_greedy`. Une des manières d'implémenter le glouton prévoyant est d'effectuer pour chacune des couleurs une copie du plateau de jeu, puis de modifier celui-ci en jouant cette couleur, en comptant le nombre de cases ainsi gagnées. Puis d'utiliser la fonction de la question 8 pour chacune des couleurs afin d'obtenir le nombre maximal de cases gagnées en ayant fixé le premier coup. On choisit ensuite le premier coup afin de maximiser ce nombre. Cet algorithme effectue 7 copies du plateau en $\Theta(n^2)$, fait sept appels à `update_board_optimized` en $\Theta(n^2)$ et

fait appel 49 fois à la fonction `calc_new_cells_optimized` qui s'exécute en $\Theta(n^2)$. Il s'exécute donc en $\Theta(n^2)$. Si on veut explorer m coups consécutifs, cette complexité passe à un $\Theta(\exp(m) * n^2)$, car il faut alors effectuer 7 fois plus de copies du tableau pour chaque coup additionnel (sauf le dernier ou il est possible d'utiliser `calc_new_cells_optimized`).

2.6 Le pire du monde merveilleux des sept couleurs (bonus)

Question 14

Cet algorithme n'a pas été implémenté.

En revanche, on remarque que l'évolution du plateau de jeu est radicalement différente entre le joueur artificiel hégémonique et le glouton. De plus, l'efficacité de cette évolution varie grandement en fonction de l'avancée du jeu. En effet, l'algorithme du glouton hégémonique va avoir tendance à étendre la zone maîtrisée par le joueur en se ramifiant énormément, quitte à laisser des "trous" dans la zone conquise. Par ailleurs, cet algorithme semble extrêmement efficace en début de partie, au contraire, la fin de partie est très lente : si la zone maitrisée englobe totalement une zone non conquise par exemple, alors conquérir les cases au bord de celle-ci réduit le périmètre, cette conquête va donc avoir tendance à être ralentie. À l'inverse la zone maitrisée par l'algorithme glouton semble être plus compacte et évoluer un peu plus lentement.

De ces observations, il paraît judicieux d'utiliser l'algorithme du glouton hégémonique jusqu'à un certain point, avant de changer de stratégie, et d'utiliser l'algorithme glouton classique ou prévoyant.

Reste à déterminer le moment judicieux pour effectuer la transition. Cela peut se déterminer expérimentalement en traçant la vitesse d'expansion de la zone maîtrisée par l'algorithme hégémonique en fonction du temps. Ou bien en traçant celle ci en fonction du pourcentage d'occupation du terrain. Ou encore en choisissant de déterminer le tour de transition à l'aide d'une loi de Poisson par exemple, dont le paramètre λ optimal, serait recherché expérimentalement.

3 Synthèse

Plusieurs joueurs artificiels ont été implémentés:

- l'algorithme glouton classique consistant à maximiser le nombre de cases gagnées à chaque tour,
- l'algorihtme glouton prévoyant, faisant le choix permettant de maximiser le nombre de cases gagnées au terme de deux tours,
- l'algorithme glouton hégémonique, maximisant le périmètre de la zone maîtrisée.

L'algorithme hégémonique est assez efficace en début de partie, à tendance a créer une zone comportant de nombreuses ramifications et trous, mais devient peu efficace en fin de partie. L'algorithme glouton est globalement plus efficace, crée une zone plus compacte.

A Annexe

A.1 Bibliographie

Ne sont cités ici que les ouvrages et sites dont nous nous sommes le plus servi.

[1] M. QUINSON : Mémoire dynamique en c. http://people.irisa.fr/Martin.Quinson/Teaching/ArcSys/, 2017. Makefile, main...

[2] WIKIPÉDIA : 7 colors. https://en.wikipedia.org/wiki/7_Colors, january 2017.