Les 7 merveilles du monde des 7 couleurs

TP noté - Architecture et Programmation C

15 octobre 2019

1 Introduction

L'objectif de ce projet est de coder *le merveilleux jeu des 7 couleurs* ainsi que des stratégies de jeux. Les règles du jeu sont rappelées ici.

Nous allons donc devoir coder un plateau de jeu interactif, qui sera initialisé aléatoirement et qui évoluera au fur et à mesure des coups des joueurs. Les joueurs pourront être des joueurs réels ou bien des stratégies que nous allons implémenter. Le but est de tester différentes stratégies de jeux, de pouvoir les comparer et d'en tirer une stratégie hybride la plus forte possible.

Ce projet nous permet d'apprendre à coder dans le langage C proprement, de faire un travail en équipe sur une durée de deux semaines mais aussi de voir comment un programme peut permettre de résoudre certains problèmes ou inspirer un humain pour choisir une stratégie de jeu.

2 Voir le monde en 7 couleurs

Question 1 La première étape est de coder le plateau de jeu qui contient la couleur de chaque case du plateau *board*, partant du template fourni. Pour ce faire nous écrivons la fonction `get_rd_color : void -> char` dont chaque appel renvoie une lettre parmi les couleurs possibles.

Pour utiliser la fonction `rand` nous avons importé la bibliothèque *time* et initialisé le temps dans la fonction `main`. Il suffit ensuite de créer la fonction `set_board : char (*)[] -> int -> void` qui choisit aléatoirement toutes les cases (sauf les cases initiales des zones des joueurs) à l'aide de la fonction précédente.

Question 2 Méthode naïve : on parcourt linéairement le monde jusqu'à trouver une case à mettre à jour (qui est de la couleur indiquée et dont une case adjacente est déjà dans la zone du joueur) et on la met dans la zone. On continue à parcourir le monde jusqu'au bout. Si on a modifié au moins une case en parcourant le monde, alors on le reparcourt. On ne s'arrête que lorsqu'on a obtenu un parcours complet sans changement.

Pour cette question, nous avons décidé de créer le tableau *adj*, qui est une matrice de taille $BOARD\_AREA$, initialisée grâce à la fonction `set_adj : void -> void` qui contient, pour chaque case, le tableau (de taille 4) des coordonnées de ses cases adjacentes. Ainsi, lors du parcours du plateau de jeu, on a directement en mémoire un accès aux cases voisines d'une case donnée et il suffit pour chaque case d'utiliser la fonction `is_available : int -> int -> int` pour savoir si elle fait partie du plateau de jeu.

Remarque : si on avait voulu créer un monde torique, il aurait suffi de changer la matrice *adj* pour relier les bords hauts et bas du tableau de jeu.

Finalement, on crée la fonction `add_cell_player : char (*)[] -> int -> int -> char -> plr -> int`. Cette fonction prend en argument un état du plateau de jeu, les coordonnées d'une case, une couleur et l'identité du joueur (représenté par un élément de la classe *plr*). Elle change la couleur de

la case si celle-ci est gagnée par le joueur et retourne ensuite un booléen (en *int*) pour indiquer si la case a été gagnée ou non.

De là, on construit la fonction `update_board : char (*board)[] -> char -> plr -> void` qui met à jour le tableau en suivant la méthode naïve précédemment décrite.

Pour vérifier que `update_board` fait bien le travail qu'on veut, il suffit d'utiliser la fonction `print_board : char (*)[] -> void` sur un plateau de jeu donné, de faire la mise à jour pour une couleur et un joueur donné et d'afficher la modification du tableau obtenue. Cette méthode est bien sûr empirique et doit être itérée sur tous les cas possibles.

Étant donné que la fonction `add_cell_player` s'exécute en temps constant et qu'une case ne peut changer de couleur qu'une seule fois (on considère qu'une case gagnée par un joueur ne redevient jamais neutre et ne peut pas être gagnée par l'autre joueur), alors chaque exécution de la fonction `update_board` est de complexité temporelle $O(BOARD\_SIZE^2)$ (double boucle for sur la largeur du plateau) d'opérations en coût constant.

Ainsi, l'un des pires cas pour cet algorithme est un tableau bicolore en serpentin (de longueur $O(BOARD\_AREA)$) où le parcours se fait en suivant le serpentin mais vers le joueur qui joue et pas en partant de celui-ci. En effet on aurait alors un seul ajout de case. On exécute donc $O(BOARD\_AREA)$ la fonction `update_board` fois ce qui nous donne une complexité totale de $O(BOARD\_SIZE^4)$. C'est peu efficace et trop lent au vu des attentes des questions suivantes (générer un grand nombre de parties).

Question 3 Afin d'améliorer notre fonction précédente, on va en réalité faire un parcours en profondeur du plateau de jeu en partant des cases déjà possédées par le joueur qui vient de choisir son coup. Pour cela on crée la fonction `explore : char (*)[] -> int -> int -> char -> plr -> int -> int`. Cette fonction permet de traiter une cellule, c'est-à-dire de la changer de couleur dans le board en argument si elle est gagnée par le joueur, de compter le nombre de cases ajoutées (grâce au compteur *counter*) et de traiter les cellules voisines. Cette fonction est utilisée plus globalement dans la fonction `search : char (*)[] -> char -> plr -> int` qui remplace `update_board` et fournit le même résultat au détail près qu'elle renvoie le nombre de cases qui ont été ajoutées.

Comme on effectue un parcours sur notre tableau, et qu'on peut faire une analogie entre ce dernier et un graphe non orienté (chaque nœud a au plus 4 arêtes, ses cases adjacentes) alors notre fonction `search` est de complexité $O(BOARD\_AREA)$.

Pour tester notre fonction, on procède de la même façon que pour tester `update_board`.

3 A la conquête du monde

Question 4 Pour faire jouer un humain contre un humain, sans s'occuper pour l'instant des conditions d'arrêt du jeu (on utilisera la commande ctrl + C du terminal pour arrêter le jeu) on procède comme suit :

La fonction `init_game : int -> char (*)[]` permet d'initialiser un plateau de jeu (bien sûr dans une variable statique) et d'en renvoyer l'adresse. Elle prend en argument un entier qui permet de savoir si le monde généré doit être aléatoire ou non (cf Q9).

La fonction `who_start : void -> plr` renvoie aléatoirement l'identité du joueur qui va commencer à jouer (pour qu'aucun joueur ne soit avantagé ; cette fonctionnalité sera notamment utile pour comparer les joueurs non humains).

L'implémentation actuelle ne prend pas en compte de condition d'arrêt donc on peut penser que notre implémentation est limitée puisqu'elle ne détermine jamais de vainqueur, d'où la question 5.

De plus, on pourrait penser que pour jouer une partie à deux humains il faut deux humains et qu'un étudiant en informatique n'a pas toujours une personne bienveillante sous la main pour s'amuser. On s'intéressera alors particulièrement aux fonctions du fichier *ai.h*.

Question 5 D'après les règles du jeu, le joueur qui gagne est celui qui possède la plus grande zone (en nombre de cases) à la fin de la partie. Mais il suffit qu'à un moment donné de la partie, l'un des deux joueurs possède au moins la moitié des cases du plateau dans sa zone pour qu'il soit déclaré vainqueur (puisqu'une case gagnée ne peut être perdue).

Cette condition est suffisante mais non nécessaire car en fait un joueur peut théoriquement être déclaré vainqueur s'il a sous contrôle la moitié des cases du plateau. Par "sous contrôle", on parle des cases qu'il possède et des cases encore neutres mais qui sont inaccessibles à l'adversaire.

La fonction `possession : char (*)[] -> plr -> int` renvoie le nombre de cases possédées par un joueur sur le plateau de jeu donné. Elle est utilisée dans la fonction `is_end : char (*)[] -> plr` qui détermine si la partie est terminée. Pour cela celle-ci renvoie l'identité du joueur qui gagne si la condition suffisante est remplie pour l'un des deux joueurs, et le joueur \0 sinon.

La fonction `play_game : (char (*)[] -> plr -> char) -> (char (*)[] -> plr -> char) -> int -> int -> plr` permet de jouer une partie en prenant en compte les différentes améliorations que nous avons effectuées par la suite. Elle prend en paramètre deux fonctions qui codent la stratégie adoptée par chacun des joueurs, la volonté (ou non) de faire un affichage (par exemple s'il n'y a pas de joueur humain, on ne veut pas forcément voir le déroulement de la partie mais seulement son résultat) et un entier (savoir si le plateau est aléatoire ou non, cf Q9). On notera qu'après avoir demandé à une stratégie son choix (i.e. la couleur qu'elle a décidé de jouer), si ce choix est '\0', alors `play_game` interprétera cela comme un abandon de cette stratégie. Ce choix d'implémentation permet à certaines stratégies d'informer le jeu qu'elles savent qu'elles ont perdu, et évite alors de continuer une partie dans laquelle seul un joueur continue à gagner des cases.

4 La stratégie de l'aléa

Question 6 Avec la fonction `get_rd_color` déjà écrite, la fonction qui joue pour un joueur artificiel en choisissant à chaque tour une couleur aléatoirement parmi les 7 couleurs du monde s'écrit facilement. Elle est notée `rand_ai : char (*)[] -> plr -> char`. Les arguments sont là pour l'homogénéité des joueurs artificiels (cf les autres implémentations).

Question 7 Comme le joueur artificiel précédant choisit parfois une couleur qui ne lui ajoute pas de case (il est donc plus aisé à l'autre joueur de gagner car cela revient à lui donner deux coups consécutifs), on ajoute au joueur précédant la spécification suivante : la couleur aléatoire qu'il choisit doit lui permettre de gagner au moins une case.

Pour ce faire, on implémentera ce joueur par la fonction `rand_available_ai : char (*)[] -> plr -> char`.

Dans un premier temps, on crée *colors_worth* qui est le tableau correspondant au nombre de cases que l'on gagne si on joue une certaine couleur (l'entier dans la case 0 du tableau correspond au choix A, et linéairement jusqu'à la case 6 qui correspond au choix G).

On choisit ensuite aléatoirement la couleur parmi les possibles (et on renvoie '\0' si aucun coup n'est possible).

5 La loi du plus fort

Question 8 On imagine à présent un joueur artificiel, `greedy_ai : char (*)[] -> plr -> char` qui, à chaque tour choisit une couleur qui lui permet d'ajouter le maximum de cases possibles à sa zone. On cherche à implémenter cette méthode car c'est plus ou moins la première méthode qu'un humain cherche à appliquer lorsqu'il découvre le jeu.

Pour cela, on utilise le même tableau *colors_worth* que pour le joueur précédent sauf que `greedy_ai` choisit la couleur grâce à la fonction `max_verifying : int ()[NB_COLORS] -> int ()[NB_COLORS] -> int` qui donne l'indice du plus grand entier d'un tableau vérifiant une propriété donnée en argument. La condition est en fait ici le même tableau, *colors_worth*, puisque ce qu'on veut c'est que le coup joué transforme au moins une case. Ainsi si `max_verifying` ne trouve pas d'indice correspondant au max, cela veut dire que toutes les couleurs n'ont aucun effet.

Question 9 On remarque après quelques parties que le joueur glouton gagne facilement et fréquemment contre le joueur aléatoire (version 2). On se demande alors s'il existe une façon d'équilibrer les parties.

Pour que la configuration soit équitable entre le joueur aléatoire (version 2) et le joueur glouton il faut que le choix aléatoire de la couleur corresponde à chaque coup à peu près à la couleur dominante dans $colors\_worth$. Pour cela on a deux possibilités :

1. Changer la loi de probabilité du joueur aléatoire pour qu'elle dépende du poids de chaque couleur (et non une loi uniforme sur les possibilités).
2. Essayer de trouver une configuration du plateau initiale qui n'avantage aucun des deux joueurs.

Pour le choix du plateau, on peut l'initialiser de façon déterministe de la façon suivante : à chaque coup, il y a une seule couleur qui peut être jouée et rapporter des cases à chaque joueur. Ce nombre de cases doit être à peu près le même à chaque coup donc on propose le plateau construit par la fonction `set_board` avec 0 comme deuxième argument : chaque anti-diagonale est d'une couleur donnée, distincte de celles des anti-diagonales voisines. Comme le joueur qui commence est nécessairement celui qui gagne mais que le choix du joueur qui commence est aléatoire, alors ce plateau permet que le combat soit équitable.

Remarque : le plateau proposé n'est pas le seul qui convient mais c'est un moyen simple de visualiser l'idée sous-jacente.

Question 10 Afin d'avoir une vue d'ensemble sur nos simulations et pouvoir faire quelques statistiques on cherche à faire un championnat (composé d'un grand nombre de parties) et donc les automatiser.

Pour cela on crée la fonction `combat : (char (*)[] -> plr -> char) -> (char (*)[] -> plr -> char) -> int -> int -> int -> void` qui utilise la fonction `play_game`. On obtient en particulier les statistiques suivantes :

Entre `rand_available_ai` et `greedy_ai` : 2-98 pour 100 parties (95 543 - 4 457 pour 100 000 parties) sur un terrain aléatoire. On obtient du 48-52 sur le terrain décrit ci-dessus. On voit donc que notre plateau permet bien d'équilibrer le combat.

Remarque : si on fait un championnat à 100 000 parties, on obtient 50 038 - 49 962 pour le plateau non-aléatoire. On voit que la répartition empirique s'approche bien du $50\% - 50\%$ théorique. Cela concorde avec nos calculs.

6 Les nombreuses huitièmes merveilles du monde (bonus)

Question 11 Une autre stratégie à laquelle un joueur humain peut penser est la stratégie hégémonique (i.e. maximiser le périmètre de sa zone). En effet, on intuite que plus on touche de cases, plus chaque coup aura d'impact. Pour implémenter cela, on va tout d'abord coder la fonction `perimeter : char (*)[] -> plr -> int` qui calcule le périmètre de la zone d'un joueur (défini comme le nombre de cases distinctes qui ne sont pas de sa couleur et qui sont adjacentes à une case de sa couleur). Ensuite, on peut procéder de la même façon qu'avec `greedy_ai` : on a un tableau $colors\_worth$ dont on veut maximiser la valeur, mais tout en respectant la contrainte $nb\_modified$ qui donne le nombre de cases modifiées par cette couleur.

Question 12 Lorsqu'on code la fonction `greedy_ai` il est naturel de chercher à l'exploiter sur plusieurs coups à l'avance. Par exemple sur deux coups. Mais comme les apprentis informaticiens que nous sommes sont gloutons par nature, on va coder directement le glouton prévoyant sur n coups pour n quelconque. Pour cela on va coder une nouvelle structure de données : un arbre. L'idée est la suivante :

— la racine de l'arbre est le plateau de jeu actuel.
— pour chaque nœud de l'arbre, ses fils représentent le choix d'une couleur à partir d'un plateau de jeu donné et le sous arbre qui lui est associé. Le $root\_letter$ est le coup joué, $root\_value$ est le nombre de case qui serait ajouté si on jouait ce coup et new_board le nouveau plateau de jeu que l'on obtiendrait. On utilise l'implémentation fils/frère pour gérer les liens entre les nœud de l'arbre.
— Si un coup n'apporte aucune nouvelle case à la zone alors on stockera un pointeur nul dans la variable *fils*.

Avec cette structure, il faut donc construire l'arbre de profondeur n. Puis en parcourant l'arbre on note le chemin qui maximise le nombre total de cases gagnées. Il suffit alors de jouer la première couleur du chemin pour son coup puis de recalculer l'arbre lors de son prochain tour.

Remarque 1 : cette structure est coûteuse en temps (recalcul de l'arbre à chaque tour) et en mémoire (on stocke un plateau de jeu par nœud) mais elle permet d'implémenter la stratégie proposée. Le calcul de l'arbre à un coup donné se fait en $O(NB\_COLORS^n * BOARD\_AREA)$ dans le pire des cas (i.e. si chaque couleur est utile sur les n prochains coups).

Remarque 2 : on comprend enfin pourquoi toutes nos fonctions précédentes prenaient en argument un pointeur vers un plateau de jeu : cela permet d'éviter de recopier le plateau de jeu un grand nombre de fois.

Remarque 3 : on est obligé de recalculer l'arbre à chaque coup car un coup de l'adversaire peut nous empêcher d'accéder à certaines zones.

Remarque 4 : pour éviter de stocker une copie de tableau à chaque nœud, on aurait pu créer une fonction qui donne le board avant d'avoir joué un certain coup.

Remarque 5 : notre implémentation d'arbre n'a pas aboutie. On laisse les `printf` pour mettre en valeur notre problème. Le commentaire associé au fichier *tree.c* explique brièvement ce qui se passe.

7 Le pire du monde merveilleux des 7 couleurs (bonus)

Question 14 On peut également créer des stratégies hybrides. La première possibilité est de fonctionner par pondération de plusieurs stratégies (pas forcément un isobarycentre). La deuxième possibilité est de définir une fonction de choix (qui prend par exemple le monde en paramètre) qui décide à chaque coup quelle stratégie est employée ce tour-ci. Cette idée est riche car il y a plein de possibilités pour cette fonction de choix : aléatoire, selon le nombre de tours écoulés... Nous avons exécuté la première en fusionnant `greedy_ai` et `hegemonic_ai`. La stratégie hybride qu'on a implémentée bat largement les stratégies précédentes. Par exemple si on fait un championnat à 100 parties contre la stratégie gloutonne, l'hybride gagne à 67-33. Contre la stratégie hégémonique, l'hybride gagne à 72-28.

8 Synthèse

Finalement, si on fait un résumé de tous les joueurs qui ont été implémentés :

- On peut améliorer la fonction qui choisit aléatoirement une couleur simplement en lui faisant choisir une case qui lui fait gagner des cases mais elle reste très faible.
- Cependant il est possible de générer des plateaux de jeu plus déterministe qui permet de rééquilibrer le combat, par exemple contre la gloutonne.
- La stratégie gloutonne est plutôt forte, et on peut encore l'améliorer en prévoyant plusieurs coups à l'avance. Nous avons eu des problèmes d'implémentation de graphes pour coder la fonction glouton prévoyant mais on s'attend à des résultats encore plus spectaculaires.
- Cependant, on peut contrer cette stratégie en essayant de la restreindre dans un domaine de moins de $BOARD\_AREA/2$ cases. En effet les zones gagnées par la stratégie gloutonne sont plutôt de faible périmètre (et de grande aire) donc sa progression sur le plateau de jeu est plus "bloc" que "filaire".
- De nos observations sur la façon dont se déroule les parties pour la stratégie hégémonique on peut dire que : cette stratégie est forte en début de partie mais plus la partie avance plus cette stratégie est faible.
- Une proposition de stratégie hybride de deuxième type serait de suivre l'hégémonique au début et la gloutonne vers la fin. C'est celle qu'on a réalisé est elle est effectivement la meilleure stratégie qu'on ait plus testée.

Idées de bonus : chercher à coder la stratégie qui vise à empêcher l'adversaire d'ajouter des cases à sa zone en l'isolant des cases neutres du jeu, pondérer les cases (par exemple dire que celles du milieu valent plus de point car permettent d'en atteindre d'avantage ensuite et l'ajouter à la fonction gloutonne prévoyante), essayer de coder un algorithme génétique sur les pondérations, faire une stratégie d'hégémonique prévoyante.