Projet 7 colors

Igor MARTAYAN et Clément MORAND

13 octobre 2020

Introduction

Le but de ce projet est d'implémenter le jeu des 7 couleurs en C, avec un code aussi clair et structuré que possible. Nous avons essayé de faire un code suffisamment générique pour qu'il soit facilement ajustable et réutilisable par la suite : cela passe par l'utilisation de structures et de méthodes générales, ainsi qu'un découpage en modules relativement indépendants les uns des autres. Au-delà du développement du jeu en tant que tel, nous nous sommes aussi beaucoup intéressés à la conception de joueurs artificiels avec différentes stratégies : aléatoire, gloutonne, hégémonique, expansionniste, minimax glouton, minimax expansionniste...

Structure du projet

Au fil des questions, nous ferons référence à plusieurs fonctions réparties dans différents modules, c'est pourquoi il peut être utile de d'expliquer rapidement la structure générale du projet pour s'y retrouver plus facilement :

- `structures.c` contient toutes les structures de données utilisées dans le programme (joueur, état du jeu, strategie, liste chaînée, file)
- `utils.c` contient plusieurs fonctions utilitaires utilisées dans les autres modules (manipulation d'une case, couleur aléatoire, condition d'arrêt)
- `display.c` contient les fonctions qui font l'affichage du jeu (plateau, score, résultats, statistiques)
- `input.c` contient les fonctions permettant à l'utilisateur d'interagir avec le programme (choix du mode, du prochain coup, nouvelle partie)
- `board.c` contient les fonctions qui initialisent et mettent à jour le plateau
- `strategies.c` contient les différentes stratégies qui peuvent être utilisées par les joueurs durant la partie
- `game.c` contient les fonctions qui permettent le déroulement d'une partie en fonction du nombre de joueurs et de la stratégie qui est utilisée par chaque joueur
- `main.c` contient le programme principal permettant de lancer une partie normale ou un tournoi

1 Voir le monde en 7 couleurs

Question 1. Le code correspondant est dans la fonction `init_board` du module `board.c`. On initialise chaque case du plateau avec une couleur aléatoire (en utilisant `rand`). Le plateau est passé dans une structure State qui nous permet de nous affranchir de variables globales. Cela nous sera utile dans la suite du projet, notamment pour implémenter des intelligences artificielles et manipuler des copies du tableau. On initialise zones de départ de chaque joueur avec `'1'` pour le premier joueur et `'2'` pour le deuxième.

Question 2. La version naïve est contenue dans la fonction `naive_update_board` du module `board.c`. On parcourt le plateau jusqu'à trouver une case de la couleur indiquée et on la modifie si l'un de ses voisins appartient au joueur. À la fin du parcours, si l'une des cases a été modifiée, on recommence la procédure. Pour vérifier le bon fonctionnement de la fonction, on effectue des tests pour différentes situations. On peut par exemple créer un plateau dont on connaît l'état après une mise à jour pour une couleur donnée et on vérifie que le plateau qu'on a mis à jour correspond bien à celui attendu. Dans le pire cas, on parcourt $\mathcal{O}(c)$ fois le plateau, où c représente le nombre de cases du tableau. La complexité totale est donc en $\mathcal{O}(c^2)$ dans le pire des cas.

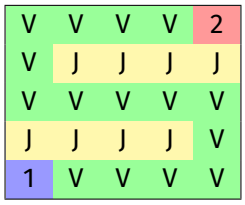

FIG. 1 : Exemple de plateau pour lequel la complexité totale est en $\mathcal{O}(c^2)$

Question 3. En premier lieu, pour obtenir une version plus efficace, nous avons décidé d'utiliser un algorithme récursif, implémenté dans la fonction `recursive_update_board` du module `board.c`. On parcourt linéairement le plateau et dès que l'on trouve une case appartenant au joueur, on appelle notre procédure récursive sur chacun de ses voisins. Notre procédure agit comme suit : si la case est de la couleur indiquée alors on la change pour l'inclure dans la zone du joueur et on appelle la procédure sur chacun de ses voisins, sinon on ne fait rien. Cela nous permet de mettre à jour le plateau en un seul parcours. La complexité totale est donc en $\mathcal{O}(c)$ dans le pire des cas.
Nous avons finalement choisi d'effectuer parcours en largeur depuis la position de départ du joueur, implémenté dans la fonction `update_board` du module `board.c`. Cette méthode a l'avantage d'être réutilisable pour plusieurs autres fonctions, notamment celles des joueurs artificiels. Pour stocker les cases à visiter, nous avons implémenté une structure de file avec deux listes chaînées (pour plus de détails, voir le module `structures.c`). Pour vérifier que cette méthode fonctionne correctement, on compare le résultat obtenu avec ces méthodes et avec la méthode précédente.

2 À la conquête du monde

Question 4. Le code permettant de lancer une partie se situe dans la fonction `run_game` du module `game.c`. Cette fonction prend en entrée l'état du jeu ainsi qu'une stratégie pour chacun des joueurs : le type `strategy` correspond à un pointeur de fonction prenant un état en entrée et renvoyant une couleur. Dans le cas d'un joueur humain, la stratégie correspond simplement à lire la couleur entrée par le joueur. Le code correspondant est dans la fonction `ask_player_move` du module `input.c`. En particulier, on veille à ce que la valeur entrée soit une couleur valide et on vide le buffer après chaque lecture. Notre implémentation pourrait être plus agréable à jouer si on sortait du terminal en utilisant une interface graphique.

Question 5. La partie est terminée lorsque l'un des joueurs possède strictement plus de la moitié des cases du plateau ou que toutes les cases sont contrôlées par les deux joueurs. Cette condition est contenue dans la fonction `game_ended` du module `utils.c`. L'affichage des scores se fait grâce à la fonction `print_score` du module `display.c`.

3 La stratégie de l'aléa

Question 6. On tire un entier aléatoire et on joue la couleur associée, le code correspondant est dans la fonction `random_color` du module `utils.c`.

Question 7. On commence par effectuer un parcours en largeur en partant de la position initiale du joueur pour trouver les couleurs atteignables. On choisit ensuite au hasard l'une des couleurs atteignables. Le code correspondant est dans la fonction `random_reachable_color` du module `strategies.c`. Cette stratégie est déjà nettement plus efficace que la précédente, on verra plus tard dans les résultats du tournoi qu'elle gagne presque systématiquement face au joueur totalement aléatoire.

4 La loi du plus fort

Question 8. Pour implémenter la stratégie gloutonne, on effectue un parcours en largeur en simulant le nombre de cases ajoutées pour chacune des couleurs et on compare les résultats pour renvoyer la couleur qui maximise l'augmentation de la taille de la zone. Une solution naïve aurait été de faire une copie du plateau pour chaque couleur et mettre à jour cette copie du plateau. Il s'avère en fait que ce n'est pas nécessaire pour simuler un seul coup d'avance.

Question 9. Pour obtenir une partie plus équitable, l'une des possibilités serait d'avoir un plateau symétrique qui donne accès aux même cases à chacun des joueurs. Cependant, le premier joueur reste grandement avantagé par le fait qu'il a toujours un coup d'avance.

Question 10. La fonction `run_fast_game` du module `game.c` permet de lancer des parties rapides (sans affichage ni délai entre chaque tour). On lance ensuite 100 parties rapides en changeant le joueur qui commence à chaque partie et en tenant compte des scores de chaque joueur pour en faire une moyenne. Sans surprise, le joueur glouton gagne l'ensemble des parties face au joueur aléatoire.

joueur / opposant	aléatoire	aléatoire+ (atteignable)	glouton
aléatoire	–	991 V / 9 D	1000 V / 0 D
aléatoire+	9 V / 991 D	–	1000 V / 0 D
glouton	0 V / 1000 D	0 V / 1000 D	–

TAB. 1 : Résultats de 1000 parties entre les joueurs artificiels

5 Les nombreuses huitièmes merveilles du monde

Question 11. Le développement de cette stratégie s'est fait en quatre phases :

- Tout d'abord, nous avons implémenté une fonction permettant de calculer le périmètre de la zone d'un joueur. Le code correspondant est dans la fonction `color_perimeter`.
- Ensuite, nous avons ensuite implémenté une version qui ne tenait pas compte du bord du plateau dans le calcul du périmètre puisque cela ne donne pas accès à des cases sur lesquelles le joueur peut s'étendre.
- Pour améliorer les résultats obtenus, nous avons ajouté une heuristique pour ne prendre en compte que les cases permettant de s'étendre par la suite (c'est-à-dire les cases ayant au moins un voisin que l'on peut colorer). On tient également compte des cases qui sont en contact avec la zone adverse car elles empêchent sa propagation.
- Enfin, nous avons mis en place une stratégie hybride entre le joueur glouton et le joueur hégémonique qui cherche à maximiser la somme du nombre de cases rajoutées et du périmètre de la zone.

joueur / opposant	glouton	hégémonique	hégémonique- (sans bord)	hybride	hybride+ (sans bord)
glouton	–	584 V / 412 D	453 V / 539 D	720 V / 276 D	772 V / 225 D
hégémonique	412 V / 584 D	–	411 V / 583 D	651 V / 343 D	687 V / 307 D
hégémonique-	539 V / 453 D	583 V / 411 D	–	704 V / 291 D	749 V / 241 D
hybride	276 V / 720 D	343 V / 651 D	291 V / 704 D	–	573 V / 415 D
hybride+	225 V / 772 D	307 V / 687 D	241 V / 749 D	415 V / 573 D	–

TAB. 2 : Résultats de 1000 parties entre les joueurs artificiels

Question 12. On remarque que le glouton prévoyant s'apparente à une implémentation de l'algorithme minimax qui utilise le score comme fonction d'évaluation et qui ne tient pas compte de l'adversaire. Pour prendre en compte les n prochains coups avec un plateau contenant c cases, la complexité est en $\mathcal{O}(7^n \times c)$. Nous avons décidé d'implémenter l'algorithme minimax puis sa version avec élagage alpha beta, qui permet de ne pas calculer certaines branches de l'arbre quand on sait qu'elles ne peuvent pas mener à un meilleur résultat, dans l'espoir de pouvoir faire jouer une version qui calcule 4 coups en profondeur en tenant compte des coups de l'adversaire.

6 Le pire du monde merveilleux des 7 couleurs

Question 13. On peut difficilement construire un monde qui rende la stratégie aléatoire inefficace. En revanche, on peut faire en sorte de contrer les joueurs gloutons et hégémoniques en imposant de faire des sacrifices pour accéder à des coups plus avantageux. (exemples ici)

Question 14. On peut implémenter une version du minimax utilisant l'heuristique hybride que nous avons développée pour le joueur hégémonique comme fonction d'évaluation.

joueur opposant	hybride+	Harpagon	Napoléon
hybride+	–	554 V / 435 D	782 V / 211 D
Harpagon	435 V / 554 D	–	735 V / 251 D
Napoléon	211 V / 782 D	251 V / 735 D	–

TAB. 3 : Résultats de 1000 parties entre les joueurs artificiels

Synthèse

Bibliographie

- *The C Programming Language, second edition*, Brian W. Kernighan & Denis M. Ritchie
- `http://web.theurbanpenguin.com/adding-color-to-your-output-from-c/`
- `https://zestedesavoir.com/tutoriels/755/le-langage-c-1/notions-avancees/les-pointeurs-de`
- `https://en.wikipedia.org/wiki/Minimax`
- `https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning`