Compte rendu : jeu des 7 couleurs

Emma Redor - Thibaut Antoine

Introduction

Le jeu des 7 couleurs est un jeu à information complète, dans lequel deux joueurs s'affrontent sur un plateau carré. Celui-ci représente le *monde merveilleux des 7 couleurs*, sur lequel les deux adversaires devront étendre leur contrôle.

Nous avons ainsi, lors de ce projet, programmé le jeu des 7 couleurs et différentes stratégies de jeu afin de les analyser et comparer leurs efficacités. Ce rapport présente également différentes idées de joueur qui n'ont pas été réalisées concrètement mais étudiées tout de même.

Notons enfin que tous les programmes de ce projet ont été écrits en `C`.

Programmation du jeu

Implémentation des règles

◇ *Question 1*

On peut noter que le déroulement du jeu dépend en grande partie de l'état initial du plateau ; ceci est dû à la nature des règles : on décide à chaque coup d'une couleur à conquérir. Ainsi, initialiser aléatoirement les couleurs des cases permet d'en avoir une répartition en moyenne équilibrée, et donc de donner lieu à des parties équilibrées (qui ne dépendent que de la stratégie employée).

Ceci a été fait en initialisant d'abord la graine aléatoire du module, puis en parcourant naïvement le plateau. Il est utile de préciser que l'initialisation de `srand` est faite une unique fois au début du programme et non pas à l'intérieur de la fonction d'initialisation du plateau. En effet, la graine d'initialisation est donnée par `time(NULL)` qui renvoie le temps international depuis 1970, et donc deux initialisations du plateau espacées d'un temps très court pouvaient en fait donner le même résultat (voir *Question 10*).

◇ *Question 2*

Trouver un invariant de boucle pour l'algorithme proposé permet de vérifier sa correction. Par exemple, l'invariant :

> Après le i-ième tour de boucle, tous les voisins de la zone du joueur, de profondeur i, et de la couleur considérée, ont été explorés.

convient, où un voisin de profondeur 0 est une case contrôlée par le joueur, et un voisin de profondeur $i > 0$ est une case non contrôlée, adjacente à un voisin de profondeur $i - 1$.

Cet invariant montre que l'algorithme fait au plus n^2 tours de boucle, où n est la taille du plateau. On constate que cette borne est atteinte (à proportionnalité près), par exemple dans le cas d'un plateau comportant une ligne monochrome serpentant dans tout le terrain pour remplir une colonne sur deux.

◇ ***Question 3***

L'observation faite à la question précédente pose un problème de complexité. En effet, un parcours de plateau est en $\mathcal{O}(n^2)$, donc la fonction précédemment créée sera elle en $\mathcal{O}(n^4)$ dans le pire des cas. C'est beaucoup trop gros, et nous pouvons faire mieux grâce à un parcours récursif du plateau.

Nous avons commencé par définir une fonction qui recouvre une zone de cases adjacentes de la même couleur par une autre. L'algorithme part d'une des cases de cette zone, la colore et explore récursivement ses voisins à condition qu'ils soient de la bonne couleur et à l'intérieur du plateau. Cette fonction a ensuite été réutilisée pour mettre à jour la zone d'un joueur après un coup, en peignant d'abord la zone qu'il contrôle par la couleur choisie, puis en repeignant par dessus la couleur du joueur. Comme cette fonction a un coût linéaire en le nombre de cases explorées, sa complexité est dans le pire des cas un $\mathcal{O}(n^2)$.
Notons que cette méthode de mise à jour du plateau n'a pas été conservée : elle a été adaptée pour mieux convenir à l'implémentation des joueurs artificiels par la suite (

Enfin, nous avons pu tester cette fonction en comparant sur quelques essais ses résultats à son homologue itérative dont nous étions sûrs de la correction.

Implémentation de la jouabilité

◇ ***Question 4***

Globalement, l'implémentation du jeu humain contre humain s'est faite par l'ajout d'une structure `player`, contenant toutes les informations nécessaires sur les joueurs : symbole, nombre de cases contrôlées, coordonnées de départ. Celle-ci était plutôt compréhensible et efficace, elle n'a pas été vraiment modifiée jusqu'à la fin.

Notre gestion de l'interaction à l'utilisateur a une première limite dans les possibilités qu'elle laisse à l'humain. En effet, par souci de généralité les joueurs peuvent décider eux mêmes du symbole avec lequel ils joueront, et peuvent donc le choisir parmi les couleurs ce qui poserait problème pour le bon déroulement du jeu. Le même problème se pose pour le choix du coup à jouer : il est possible de sélectionner une couleur invalide, par exemple celle du joueur adverse ; comme dit le célèbre adage : *« Don't trust user input »*. Des mesures auraient pu être prises pour empêcher ces abus, mais cela n'a pas été fait par souci de temps et de pertinence dans le cadre du projet.

De même, une interface graphique plus jolie aurait pu être implémentée, que ce soit en affichant de vraies couleurs plutôt que des lettres, ou en réécrivant dans la console sur ce qui a déjà été écrit, mais encore une fois cela n'a pas été fait pour les mêmes raisons.

◇ ***Question 5***

À partir du moment où l'un des deux joueurs contrôle strictement plus de la moitié des cases du plateau, il gagne puisque son adversaire ne pourra jamais contrôler plus de cases que lui (rien n'est prévu dans les règles pour faire perdre du contrôle à un joueur). La condition d'arrêt s'écrit donc

$$\exists i \in \{1,2\}/N_i > \left\lfloor \frac{n^2}{2} \right\rfloor + 1$$

avec N_i le nombre de cases contrôlées par le joueur i.

Implémentation de joueurs artificiels

La gestion des joueurs artificiels a été faite en ajoutant un attribut à la structure `player`, appelée `ai_type` et donnant l'information sur le type de joueur artificiel qu'est l'instance considérée (l'humain est considéré comme un type de JA particulier).

Pour la gestion des coups, une fonction générale prenant en entrée une instance de `player` et retournant une couleur a été créée, celle-ci faisant un `switch case` sur `player -> ai_type`. Programmer un JA est donc revenu à coder une fonction implémentant sa stratégie.

Stratégie aléatoire

◇ ***Questions 6, 7***

L'implémentation de l'aléa naïf a été très facile : il suffisait de tirer une lettre au hasard. Néanmoins, celle de l'aléa améliorée a été un peu plus ardue puisqu'il était nécessaire de simuler un coup choisi au hasard pour vérifier qu'il apportait bien un nombre strictement positif de cases au joueur, or c'était impossible à ce moment là.

Ainsi a été modifiée l'implémentation du plateau. En effet, celle en place à ce moment ne permettait pas de simuler un coup sans le modifier : dans le cas contraire la fonction de propagation des couleurs explorerait des cases déjà explorées et donc ne terminerait pas.
Nous avons donc décidé de créer une structure pour les cases, ayant notamment pour attributs sa couleur, et un *flag*, c'est à dire un entier permettant de garder en mémoire si on l'a déjà explorée ou non. La fonction de propagation a donc été adaptée en conséquence et une fonction similaire permettant de simuler des coups sur le plateau a été crée, nous avons donc pu mettre en place la stratégie aléatoire améliorée.

Stratégie gloutonne

◇ ***Question 8***

La stratégie aléatoire étant extrêmement naïve, on peut avoir envie de trouver une stratégie un peu plus réfléchie. La première qui peut venir en tête est la gloutonne : à chaque coup, on choisit la couleur qui nous rapporte le plus de cases.

La majeure partie du travail nécessaire à l'implémentation de la stratégie gloutonne a été faite dans la question précédente lorsque la fonction de simulation a été mise en place. En effet, il a alors

suffi de calculer pour chacun des coups possibles le nombre de cases qu'il rapporterait au joueur s'il était joué, et de choisir le coup qui en faisait gagner le plus.

C'était d'ailleurs l'une des motivations pour modifier la structure du plateau : comme cela sera montré dans la suite de ce rapport, toutes les stratégies suivantes ont nécessité la simulation d'un, voire plusieurs coups dans le futur.

◇ ***Question 9***

Comme cela a été souligné au début de ce rapport, jouer sur un plateau aléatoire permet de faire en sorte qu'en moyenne l'un des deux joueurs ne soit pas avantagé par rapport à l'autre. En effet, la probabilité de générer le même plateau un grand nombre de fois étant très faible, on peut supposer qu'en général deux configurations initiales aléatoires auront un certain degré de différence entre elles.
On remarque effectivement que le nombre de couleurs identiques entre deux plateaux tiré aléatoirement suit une loi binomiale, et auront donc en moyenne n^2/k cases de couleur identique (où n est la taille du plateau et k le nombre de couleurs). Ainsi, si beaucoup de configurations initiales avantagent l'un des joueurs en particulier, on peut conclure que sa stratégie est simplement meilleur. C'est pour cette raison qu'il a été choisi de déterminer la configuration du plateau aléatoirement avant chaque match.

◇ ***Question 10***

Comme on pouvait s'y attendre, le joueur glouton gagne dans presque 100% des cas à chaque foid qu'il affronte le joueur aléatoire. On en conclut donc que cette stratégie est viable, car performe mieux que l'aléatoire.

Stratégies avancées

◇ ***Question 11***

Une autre idée de stratégie que l'ont peut avoir est de chercher à maximiser le périmètre au lieu du nombre de cases contrôlées. L'intuition est de se dire qu'en maximisant le périmètre, un joueur pourra être en contact avec plus de cases à contrôler, ce qui peut donner lieu à des possibilités de contrôle plus grandes dans les coups futurs, qui pourront peut-être eux aussi augmenter le périmètre.

Cette stratégie a été mise en place sur le même principe que la stratégie gloutonne : à chaque coup, on simule tous les choix de couleur possible et on cherche le plus avantageux. Néanmoins le joueur hégémonique se distingue du glouton en cela qu'il va chercher à toujours faire grandir son périmètre et non son aire : nous avons donc implémenté une fonction de calcul différente. Il est à noter qu'une case isolée dans la zone d'un joueur ne compte pas pour son périmètre, le contraire pourrait mener à des situations où l'hégémonique refuse de jouer de bons coups forts mais qui diminueraient trop son périmètre car permettraient de contrôler des cases à l'intérieur de sa zone.

Néanmoins, il s'est avéré que le joueur hégémonique perdait environ 75% du temps contre le glouton. Après de plus amples observations, nous avons pu remarquer que parfois, en privilégiant le périmètre de sa zone, l'hégémonique ne jouait pas certains coups qui auraient pu lui rapporter

un grand nombre de cases. Sa progression finissait alors inévitablement par ralentir, au profit de son adversaire. Cela peut mener sur la piste d'une amélioration possible de cette stratégie.

◇ ***Question 12***

La stratégie du glouton prévoyant émerge naturellement lorsque l'on souhaite améliorer la stratégie gloutonne : quitte à anticiper, autant le faire sur plusieurs coups.

Nous l'avons implémenté en nous servant des attributs *flags* des cases du plateau : en plus de pouvoir garder en mémoire quelles cases ont été explorées lors de la simulation d'un coup, on peut aussi garder en mémoire le coup lors duquel elles ont été explorées. En plus de faciliter le nettoyage de ces *flags*, cette technique permet de simuler plusieurs coups dans le futur, et c'est ce dont nous nous sommes servis pour programmer le glouton prévoyant (en appliquant la simulation au cas particulier de profondeur 2).

Néanmoins la performance de la stratégie a un coût plus élevé en complexité. En effet, comme on a pu le voir lors de la question 3, notre fonction de parcours a une complexité en $\mathcal{O}(n^2)$, et on effectue en tout k^p parcours du plateau (pour un glouton de profondeur p). Le glouton prévoyant a donc une complexité en $\mathcal{O}(k^p n^2)$. Si l'on ne considère que la taille du plateau comme paramètre, l'algorithme est toujours quadratique. Néanmoins, le coefficient de cette complexité grossit exponentiellement en fonction de la profondeur de calcul, et calculer trop loin peut potentiellement amener à un ralentissement du programme.

◇ ***Question 13***

/

◇ ***Question 14***

Un tel joueur n'a pas été implémenté, mais nous aurions pu le faire en nous servant de l'observation faite à la question 11. En effet, s'il on arrive à déterminer le moment à partir duquel il vaut mieux opter pour une stratégie gloutonne qu'une stratégie hégémonique, nous pourrions créer une stratégie hybride, prenant le meilleur de ces deux joueurs.

Ce moment pourrait par exemple être déterminé statistiquement en observant sa valeur sur plusieurs parties, lorsqu'on change la taille du plateau ou le nombre de couleurs. Une analyse plus fine des stratégies permettrait également de connaître le périmètre moyen de la zone d'un joueur hégémonique en fonction du temps sur un plateau initialisé aléatoirement.

Cette stratégie n'a néanmoins pas été implémentée par souci de temps, mais explorer son idée plus en détails reste intéressant.

Synthèse et conclusion

Concrètement, nous avons pu lors de ce projet explorer les possibilités du jeu des 7 couleurs, ainsi que différentes stratégies de jeu qui pouvaient être employées pour gagner le plus souvent possible, comme par exemple les stratégie hégémonique et gloutonne ainsi que leurs versions profondes, ou encore un hybride des deux. Nous aurions pu, au delà des stratégies exposées dans ce

projet, en implémenter d'autres, comme par exemple une stratégie *minimax* munie d'un élagage $\alpha - \beta$ se reposant sur une structure de données récursive modélisant des arbres n-aires.

Avec un peu plus de recul, ce projet nous aura surtout appris à maîtriser la notion de structure en `C`, à bien utiliser les fonctions `malloc`, `free`, etc. L'organisation du code et la communication ont été également deux points cruciaux de ce projet qui s'est fait en binôme. Enfin, d'un point de vue plus global, ce projet a donné un bon aperçu de ce à quoi nous pourrions nous attendre en terme de travail durant le reste de l'année, et fut dans tous les cas une expérience très enrichissante.

Bibliographie

— Les cours de `C` de M. Quinson