ENSEIRB-MATMECA

Projet S6

Filière Informatique

Gomoku Bitboards

Auteurs :
Marc Duclusaud
Sonia Ouedraogo
Pierre Pavia
Lucas Trocherie

Superviseur :
M. Georges Eyrolles

3 mai 2019

Table des matières

Introduction

Dans le cadre de ce projet, nous réalisons le système de jeu Gomoku qui consiste à faire jouer à tour de rôle deux joueurs ayant chacun des pions d'une couleur propre à chacun. Gomoku se joue sur un plateau à nombre de cases variables de taille $N \times N$. Les règles du jeu sont assez simples : à tour de rôle chaque joueur place son pion sur une case du plateau, le but étant d'aligner une suite d'au moins cinq pions de sa couleur pour être désigné vainqueur. L'objectif du projet est donc de mettre en place le système de jeu et d'implémenter differentes stratégies pour nos joueurs.

Ce rapport reprend en détail le travail qui a été réalisé tout en expliquant les choix d'implémentation qui ont été faits et la conception algorithmique. Nous analysons également les problèmes rencontrés et discutons de la complexité et de la correction de nos algorithmes.

Cadre de travail

La réalisation de ce projet s'est étalée sur huit semaines à raison de quatre heures par semaine sous la tutelle d'un encadrant, auxquelles il faut ajouter le temps de pratique libre non encadrée. Pendant les séances nous avons chercher à subdiviser le travail et ainsi réaliser des fonctions auxiliaires sur lesquelles nous nous appuierions pour le projet global.

Ce projet est réalisé entièrement en langage C et nous avions aussi à notre disposition une plateforme en ligne (git) où nous pouvions mettre en commun le travail de toute l'équipe. Par une série de tests la forge git nous permettait aussi de voir où nous en étions et si nous validions au fur et à mesure le travail demandé.

1 Problématiques et difficultés

1.1 Problématique du projet

La réalisation de ce projet se divise en trois parties principales. D'une part la création du plateau de jeu que nous appelerons *board* dans la suite. Il fallait en effet implémenter le board et les differentes fonctions qui ont attrait à celui ci, à savoir par exemple la fonction de placement d'un pion, la détermination des pions alignés... puis l'implémentation du joueur et de ses stratégies de jeu. Afin de gérer les parties, il fallait également réaliser un serveur de jeu.

1.2 Difficultés rencontrées

Ces différentes étapes du travail ont été réalisées en parallèle par les membres de l'équipe, et pour assurer cela il a fallu au préalabe s'accorder sur le choix des structures de données. Il fallait en effet en premier lieu déterminer des structures de données adaptés au jeu pour représenter, par exemple, la table de jeu, mais aussi le joueur. Il a fallu ensuite construire la boucle de jeu tout en tenant compte des données d'entrés qui peuvent varier, comme la taille de la table de jeu ou encore le mode swap du jeu, puis gérer l'interaction entre les joueurs, assurer la validité des coups, et détecter correctement une fin de partie. Par ailleurs, ce projet ayant génerer beaucoup de fichiers, il fallait veiller à l'organisation interne de ceux-ci et assurer le moins de modifications possibles lors de modfications postérieures d'une des structures.

2 Solutions apportées

Les contraintes évoquées dans le paragraphe précédent nous ont donc menés à faire des choix d'implémentations que nous détaillerons dans cette partie, après laquelle nous analyserons la conception algorithmique des différentes fonctions.

2.1 Organisation des fichiers

Nous aborderons dans cette partie la manière dont les fichiers sources ont été organisé et donc la manière dont le travail a été réparti. La base du projet

est composé de deux fichiers headers *move.h* et *player.h* qui font le lien entre le serveur et un joueur, et leur permettent de communiquer. La modification de ces fichiers entrainerait un refactoring de tout le projet. Le projet est ensuite scindé en trois grandes partie principales, le board, le serveur et le joueur.

2.1.1 Le board

Le board est réalisé à partir d'un fichier source qui permet de le manipuler. Son code ne dépend que du header *move.h*, autrement dit il n'a besoin de connaître que la structure de données utilisée par le player et le serveur pour communiquer leur coups, car le serveur envoie cette structure au board pour le mettre à jour. Le bitboard est construit à l'identique.

2.1.2 Le serveur

Le serveur est composé de différents fichiers sources. Il est d'abord composé des fichiers *server.c* et *server_functions.c* qui mettent en place la boucle de jeu. Le serveur est ensuite composé du fichier *game.c*, c'est ce fichier qui contient les règles du jeu et particulièrement la condition de victoire d'un joueur.

2.1.3 Le joueur

Le joueur, quel que soit sa stratégie est composé d'un ou plusieurs fichiers sources qui dépendent uniquement de *player.h* en ce qui concerne la stratégie, qui dépend lui-même de *move.h*. Cependant, le joueur ayant besoin de stocker les coups ayant déjà été joués, celui-ci a besoin d'une structure de données qui lui est propre pour ce faire, et nous avons retenu deux choix dans leur implémentation. Le player random chosit de stocker tout les mouvements qui ont déjà été joués dans un tableau, qu'il parcourt à chaque fois qu'il joue pour donner un coup valide. Le player avec une stratégie de Min-max et le player avec une stratégie Alpha-Bêta utilisent quant à eux un board comme le serveur, de manière à optimiser la complexité temporelle du jeu.

2.1.4 Organisation générale

La hiérarchie générale des fichiers peut être résumée dans le graphe 1 :

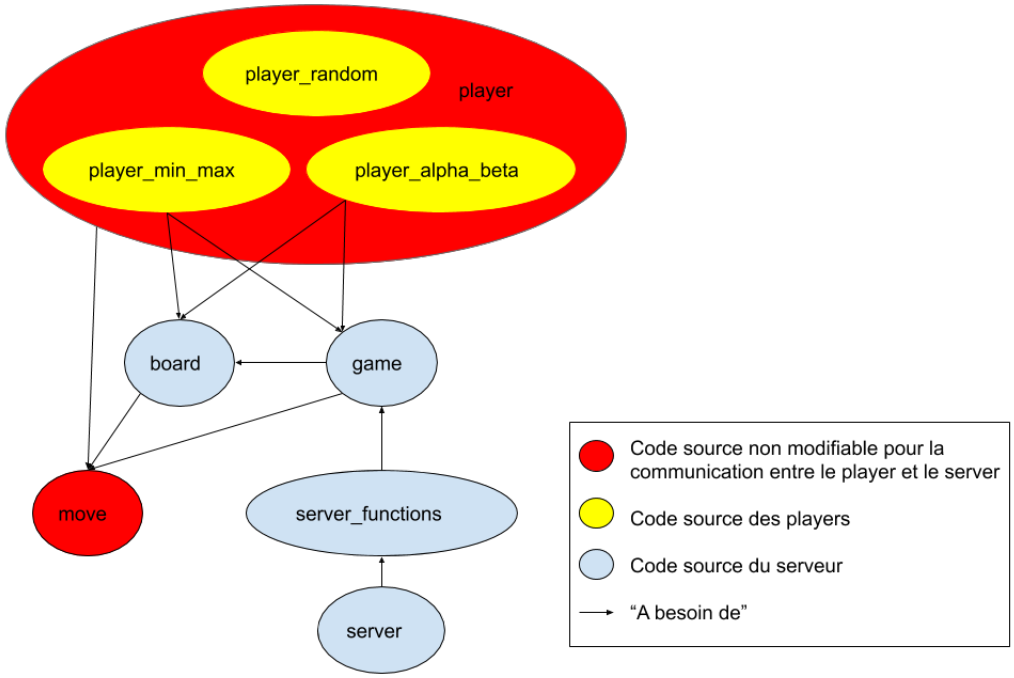

FIGURE 1 – Graphe illustrant la hiérarchie des fichiers

2.2 Les structures de données

2.3 La table de jeu

2.3.1 Prise en compte de la taille variable

Nous avions deux possibilté d'implémentation du board pour permettre la variation de la taille. Il était possible de faire une allocation dynamique qui crée le board sur commande. La deuxième possibilité, celle sur laquelle nous nous sommes penchée est d'utiliser une board déjà créée avec une taille MAX-SIZE et nous réduisons la taille sur commande. L'inconvénient est que il n'est pas possible de jouer sur toutes tailles possible à moins de relancer l'éxécutable en changeant MAX-SIZE. Mais étant donné qu'il nous était spécifié que le jeu se ferait sur une taille de board allant de 5 à 11 et étant donné la nature du jeu, nous n'avons pas trouvé intéressant d'utiliser l'allocation

dynamique.

2.3.2 Une première version basique

Une première implémentation simple et naturelle du board est de le représenter par une matrice A de taille nxn, avec n la taille de la table de jeu. La matrice est initialisée avec des -1 et le coefficient $a_{i,j}$ est ensuite modifiée par la couleur d'un joueur, représentée par un entier.

2.3.3 Le Bitboard

Une manière d'optimiser le programme en espace est de changer la manière de stocker la table de jeu, en passant d'une matrice à un bitboard. Le principe du bitboard est simple, mais il est plus difficile à manipuler. Il consiste à stocker la table de jeu sur un mot binaire, un entier de 128 bits ($\_\_int128$) dans le cadre de ce projet. Chaque bit représente une case du board, mis à 1 si la case est occupée et à 0 sinon. Il faut donc que chaque joueur ait son propre bitboard, regroupé dans un tableau dans notre implémentation. La manipulation du bitboard est réalisée avec des opérations binaires, qui posèrent problème lors de la réalisation de cette structure de données.

2.3.4 La fonction d'alignement

La fonction d'alignement compte le nombre maximum de pions alignés pour un joueur, de manière à décider par la suite si la condition de victoire est vérifiée. Cette fonction compte le nombre de pions dans les quatres directions de l'espace, comme montré sur la figure 2, à partir du dernier coup joué :

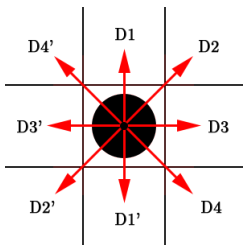

FIGURE 2 – Vérification du nombre de points alignés dans les quatres directions.

La fonction compte le nombre de pions alignés sur les directions horizontale (cf. figure 3), verticale, Sud Ouest(SO)-Nord Est(NE) (cf. figure 4) et Sud Est(SE)-Nord Ouest (NO).

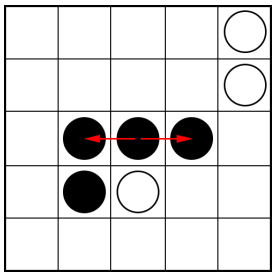

FIGURE 3 – Comptage du nombre de pions sur la direction horizontale

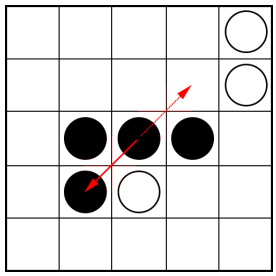

FIGURE 4 – Comptage du nombre de pions sur la direction SO-NE

Pour chaque direction, cette fonction compte le nombre de pions alignés dans les deux sens et dans les limites du damier en partant du dernier pion placé, et elle arrête de compter dans un sens lorsque elle atteint la limite du board ou qu'elle rencontre un pion d'une autre couleur que celle du joueur qu'elle traite ou une case vide.

Dans le pire des cas correspondant à la situation où les quatres directions sont remplies de pions de la même couleur que celles du joueur traité, la complexité de la fonction est $O(4n)$, où n est la taille du board.

2.4 Le joueur

2.4.1 Le joueur random

Dans un premier temps nous avons réalisé un joueur *basique*. En effet ce joueur ne posséde aucune stratégie, ses coups sont réalisés de manière aléatoire. Ce joueur essaye simplement d'aller le plus loin possible dans la partie sans faire de faute de jeu. Il choisit une position aléatoire sur le board et regarde si la place est déjà prise. Si non, il y place son pion, si oui, il tire une autre position au hasard. Nous nous sommes rendu compte très rapidement qu'un tel joueur n'était pas performant. Nous avons donc exploré des stratégies pour améliorer nos joueurs.

2.4.2 Stratégie du Min-Max

Principe

La première stratégie étudiée est le Min-Max. Le but de cette stratégie pour un joueur donné est de minimiser ses chances de perdre et maximiser celles de son adversaire. Le principe est le suivant : Avant de placer son pion, un joueur min-max simule le jeu final pour chacune des possibilité qu'il a pour placer son pion. Ainsi il place son pion en fonction de la situation finale. Cette situation finale doit donc être évaluée pour permettre au joueur de faire correctement son choix.

Fonction d'évaluation L'efficacité de tout algorithme Min-Max repose sur la fonction d'évaluation utilisée. Dans notre cas, nous avons choisi dans un premier temps une fonction d'évaluation qui note la situation finale comme indiqué plus haut. En effet :

- si avec le pion placé, le jeu simulé abouti à une victoire pour notre joueur, ce pion est noté 1000
- si c'est l'adversaire qui gagne ce pion est évalué -1000
- si la partie aboutie sur un match null, le pion est évalué 0

Nous avons choisi de modifier quelque peu cette fonction d'évaluation en multipliant par un facteur Nb_{coups} quand l'adversaire gagne et par $\frac{1}{Nb_{coups}}$ quand le joueur gagne. Nb_{coups} est le nombre de coups après laquelle la partie se termine. Ainsi le pion choisi étant celui de poids le plus élévé, nous assurons ainsi de maximiser les chances de perdre de l'adversaire et minimiser celles du joueur en lui permettant de gagner le plus tôt possible.

Implémentation

L'algorithme Min-Max est principalement basé sur de la récursivité. Nous avons implémenté deux fonction distinctes Min() et Max(). Ces deux fonctions prennent les mêmes paramètres et l'action n'est pas la même. Ces deux fonctions s'appellent récursivement.

Quand la fonction Min() est appelée, c'est notre joueur qui vient de jouer. Cette fonction commence par évaluer le pion précédemment placé et retourne une valeur si la partie arrive à sa fin. Dans le cas contraire, elle simule un pion de l'adversaire puis appelle la fonction Max() pour chacune des positions possibles sur le board. La valeur que retourne la fonction Min() est donc le retour de la fonction d'évaluation ou le minimum des valeurs renvoyées par Max().

Ainsi quand la fonction Max() est appelée, c'est l'adversaire qui vient de jouer. Le pion est évalué et la fonction s'arrête si c'est un tout final. Dans le cas contraire, on simule un coup de notre joueur pour toutes les positions restantes puis on appelle la fonction Min() sur chacune des simulation. Par le même raisonnement la valeur de retour du Max() est soit l'évaluation du pion précédent soit le maximum des valeurs renvoyées par Min().

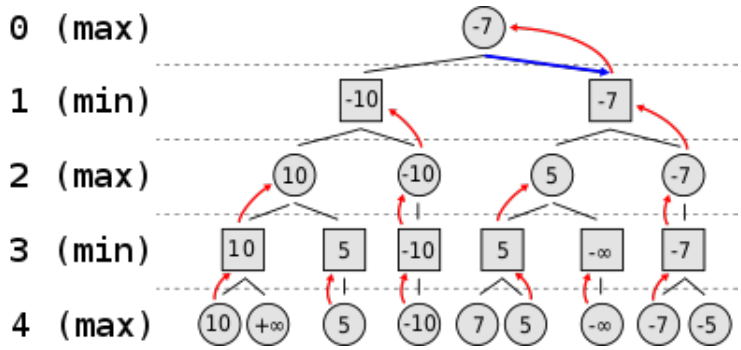

FIGURE 5 – Schéma illustrant le principe Min-Max *Source Wikipédia*

On peut modéliser la techinque par un arbre comme sur la figure 5 dont les feuilles correspondent à nos situations finales. Ainsi, récursivement on descend jusqu'à une certaine profondeur pour remonter avec la *meilleure* évaluation.

Efficacité et limites

Nous avons testé notre joueur localement avec nos joueurs random. Face à un tel joueur le min-max remporte à plate couture. Egalement quand nous faisons jouer deux players min-max avec la même stratégie, on constate que ces deux se bloquent mutuellement ce qui est normal.

Une difficulté qui est vite survenue avec cette façon de faire est le temps d'attente. Evaluer tous les noeuds de l'arbre n'était pas forcément une bonne idée, car même sur un tableau de taille réduit 5×5 le temps d'attente était très long. Non seulement le temps d'attente était long, mais la complexité de l'algorithme était exponentielle. En effet pour simplifier, supposons constante la complexité de la fonction d'évaluation. Dans le pire des cas, la hauteur de l'arbre est de n-1 avec n la taille du board. La complexité est donc en $\mathcal{O}(n^{n-1})$. Pour palier à cela nous avons donc choisi de réduire la profondeur

de l'arbre étudiée. Nous avons limiter le parcours à deux appels récursif de la fonction min() ce qui correspond à une profondeur de 4 pour l'arbre de jeu. Cela permet de réduire la complexité, elle sera polynomiale en $\mathcal{O}(n^4)$. Ainsi, si au bout de cette profondeur de jeu, on n'atteint pas des feuilles, la fonction d'évalution renvoit 0. Cette amélioration a permis de faire jouer des players, même sur des boards de taille 11×11, en moins de 5 secondes.
Il est encore possible de réduire cette complexité en utilisant un élagage alpha béta.

2.4.3 Amélioration Alpha-Béta

La stratégie alpha-béta vient en complément de la stratégie min max. Dans la dernière partie nous avons vu l'importance, et même la nécessité, de réduire la profondeur de l'arbre. Cependant le parcours en largeur restait le même. Dans l'élagage alpha béta, ce parcours en largeur est lui aussi réduit.

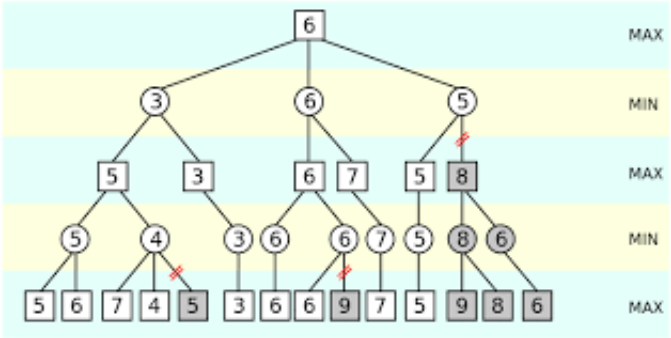

FIGURE 6 – Schéma illustrant le principe Alpha-Béta *(Source : Jez9999, Wikimedia Commons)*

Comme l'illustre la 6 il existe des noeuds de l'arbre dont l'évaluation n'est pas intéressante. Prenons le cas de la deuxième ligne qui correspond au tour du min. Comme rappelé précédemment, la fonction Min() fera appel à la fonction Max(). Il est possible en fait d'ignorer l'appel à certains appels de la fonction Max(). La règle est la suivante, on note alpha la valeur renvoyée par Max précédent l'appel à Min. Lors de l'évaluation des fils de min, dès que l'on obtient une valeur inférieur à alpha, on arrête l'évaluation des noeuds frères. Notons par exemple le cas de l'exemple après l'évaluation du premier noeud
alpha = 3.

— Lors de l'évaluation du deuxième noeud, le premier noeud fils est 6, supérieur à alpha, on continue alors normalement l'évaluation des nœuds frères et alpha prend la valeur du maximum (6 dans ce cas).
— Ensuite lors de l'évalution du noeud suivant, on tombe sur le fils 5 qui est plus petit que la valeur courante de alpha. On arrête donc l'évaluation des noeuds frères.

2.5 Le serveur

Le serveur de jeu a pour but la gestion entière du jeu. Il doit initialiser le plateau et initialiser les joueurs. Toute action qui a lieu par exemple sur le plateau de jeu est réalisée par le serveur, les joueurs ne communiquent pas directement entre eux ni avec le plateau. Le serveur assure donc cette communication.

2.5.1 Le chargement dynamique des joueurs

Il existait deux possibilités pour faire jouer les joueurs avec le serveur.
D'une part il y'a le chargment statique à travers les fichiers d'entête .h ; c'est à dire que le lien serveur/joueur sera fait dès la précompilation/compilation.
D'autre part il y'avait le chargement dynamique. Celle ci consiste à réaliser une bibliothèque dynamique avec le fichier source du joueur. Ainsi le lien serveur/joueur sera fait lors de l'exécution. Nous avons préféré la deuxième méthode. Le chargement dynamique a l'avantage de pouvoir lancer une partie sans avoir à recompiler tout le projet lors d'un changement de joueur. De plus avec le chargement statique il aurait fallu un fichier spécifique à chaque joueur avec des noms de fonctions différentes, ce problème est évité avec le chargement dynamique.
Les bibliothèque dynamiques sont chargées en début de partie et libérées en fin de partie par le serveur.

2.5.2 Prise en compte des données d'entrée variables

En effet trois éléments variables étaient à prendre en compte.
— Les bibliothèques des joueurs
— La taille du board
— Le mode de jeu : le mode standard qui represente un jeu normal et le mode swap qui consiste à faire jouer 3 coups par le premier joueur

puis le faire valider ou non par le deuxième.
Ainsi avant de commencer le jeu, le serveur doit traiter ces différentes options. Nous réalisons donc un parse-opt afin de recupérer ces différents éléments. Un traitement est alors réalisé par la fonction *error-management* qui s'assure que les données fournies sont exactes : que les bibliothèques fournies existent et la taille est bien fournie et strictement positive.

2.5.3 La communication avec les clients

Elle est entièrement gérée par le serveur. Comme spécifié plus haut, les joueurs n'ont pas accès au board, cela évite certains cas de tricherie. Ainsi ils fournissent leur coups au serveur qui les place. Pour connaître l'état du board, le serveur devait également fournir au joueur le tableau des derniers pions placé depuis son jeu. Il fallait donc veiller à ce que le joueur retienne de ces côté ces pions. Ceci était indispensable notamment pour le player-random pour lui permettre de jouer des coups valides.

2.6 Implémentation des règles du jeu

2.6.1 Détermination de coup valide

Un coup valide est un coup qui se situe à l'intérieur du board, qui ne vient pas se placer sur une case déjà occupée et qui respecte bien la couleur du player. Cette vérification est assurée par la fonction 1

Listing 1 – Fonction de vérification de la validité d'un coup

```
1  int test_valid(struct board bd,
2                 struct col_move_t cm,
3                 enum color color)
4  {
5    return( bd[cm.m.row, cm.m.col] == -1 &&
6            cm.m.col > 0 &&
7            cm.m.col < bd.size &&
8            cm.m.row > 0 &&
9            cm.m.row < bd.size &&
10           cm.c == color)
11 }
```

2.6.2 Détermination d'une situation gagnante

Une situation gagnante arrive lorsqu'un joueur aligne au moins cinq de ses pions. La fonction *is-winning* utlise ainsi une des fonction de notre TAD board qu'est *align*. L'algorithme de victoire est donc de la forme :

```
1 int is_winning(struct board bd, struct col_move_t cm)
2 {
3   int res = align(bd,cm);
4   if (res >= 5)
5     return cm.c;
6   return -1;
7 }
```

3 Améliorations

D'autres améliorations auraient pu être apportées à ce projet de Gomoku, comme :

— Avoir un bitboard fonctionnel à la place d'une matrice.

— Une stratégie de Montecarlo aurait pu être implémentée pour le joueur ou d'autres stratégies encore plus avancées.

4 Conclusion

Pour conclure, ce projet nous a permit de réaliser un jeu de Gomoku et à travers ce jeu de développer nos compétences en C en utilisant des bibliothèques dynamiques et en metant en place un serveur et des joueurs. Ce projet nous a également permit d'approcher des objets mathématiques plus théorique avec les stratégies des joueurs, notamment des arbres.