Compte rendu du projet de Programmation Avancée

Jeu de morpion client-serveur en C++

Baran CELIK 141

Table des matières

I. Introduction

Dans le cadre du cours de programmation avancée, j'ai réalisé un projet de développement d'un **jeu de morpion** en C++ reposant sur une architecture client-serveur. Le joueur peut affronter un bot selon deux niveaux de difficulté.

Ce projet utilise les sockets TCP pour établir la communication entre le client et le serveur, et les threads pour permettre au serveur de gérer plusieurs connexions en parallèle. Tout se déroule dans le terminal, en ligne de commande.

Deux types de comportements ont été implémentés pour l'adversaire robot :

- *Une stratégie naïve*, où les coups de l'adversaire sont choisis de manière entièrement aléatoire parmi les cases libres.
- *L'algorithme Minimax*, qui constitue une approche classique d'intelligence artificielle pour les jeux à somme nulle. Cet algorithme explore récursivement tous les coups possibles pour déterminer la meilleure décision à prendre, garantissant un jeu optimal de la part de l'IA.

II. Architecture générale

Le projet est structuré autour d'une architecture client-serveur, dans laquelle un client se connecte à un serveur afin de participer à une partie de morpion. Cette organisation repose sur l'utilisation de sockets TCP et sur l'emploi de threads côté serveur pour permettre la gestion simultanée de plusieurs clients.

Quatre fichiers principaux composent l'ensemble du projet : **client.cpp**, **server.cpp**, **rules.h** et **Makefile**.

- *Le fichier client.cpp* contient le programme exécuté par le joueur. Il est responsable de la connexion au serveur, de l'affichage des messages reçus, de la saisie des coups joués, et de la transmission de ces informations au serveur. Il gère également l'affichage des grilles de jeu et les messages de fin de partie.
- *Le fichier server.cpp* correspond au cœur du jeu. Il initialise un socket d'écoute et attend les connexions entrantes. À chaque nouvelle connexion, un thread est lancé pour gérer la partie associée au client connecté. Ce fichier contient la logique de déroulement du jeu, le choix du mode de difficulté, la gestion du tour par tour, et l'envoi des mises à jour de la grille au client.
- *Le fichier rules.h* centralise toutes les règles du jeu et les fonctions associées à l'intelligence artificielle. Il permet de vérifier la validité d'un coup, de détecter un gagnant ou une égalité, et de générer les coups de l'IA. Ce fichier contient deux stratégies de jeu pour l'adversaire (le bot) : une fonction de mouvement aléatoire pour le mode facile, et l'implémentation de l'algorithme Minimax pour le mode difficile. Il contient également des fonctions pour afficher la grille ou la formater sous forme de chaîne de caractères transmissible au client.
- *Le fichier Makefile* permet de compiler rapidement les exécutables du serveur et du client à l'aide d'une simple commande make. Utiliser Makefile facilite également le nettoyage des fichiers objets générés avec make clean.

III. Fonctionnement du jeu

Dans ce projet, le déroulement d'une partie de morpion suit un protocole bien défini entre le client et le serveur, basé sur des échanges de messages via sockets TCP. Le process démarre lorsque le client, lancé par l'utilisateur, initie une connexion au serveur en spécifiant son adresse IP/localhost. Une fois la connexion établie, la session de jeu peut commencer. Le serveur commence par envoyer un message au client pour lui demander de choisir le niveau de difficulté du bot. L'utilisateur doit alors entrer un entier (1 ou 2) correspondant au mode souhaité : le mode facile pour un adversaire jouant aléatoirement, ou le mode difficile pour un adversaire jouant de manière optimale grâce à l'algorithme Minimax. Ce choix est transmis au serveur, qui adapte ensuite son comportement en conséquence.

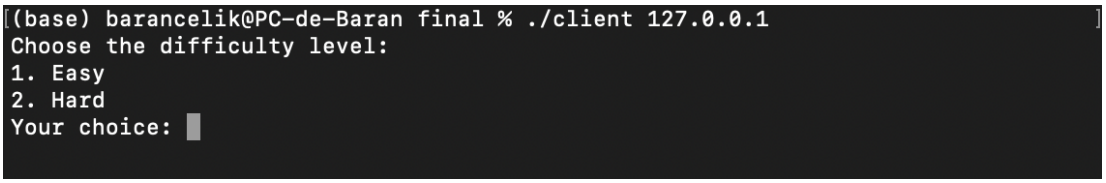

Figure 1 - Choix du niveau de difficulté : 1 pour facile ou 2 pour difficile

La partie se joue au tour par tour. Le joueur humain, qui joue toujours en premier, reçoit une grille vide au début du jeu, formatée de manière lisible dans le terminal. Il est invité à entrer les coordonnées de la case dans laquelle il souhaite jouer. Ces coordonnées sont envoyées au serveur, qui vérifie leur validité grâce à la fonction valid() définie dans le fichier rules.h.

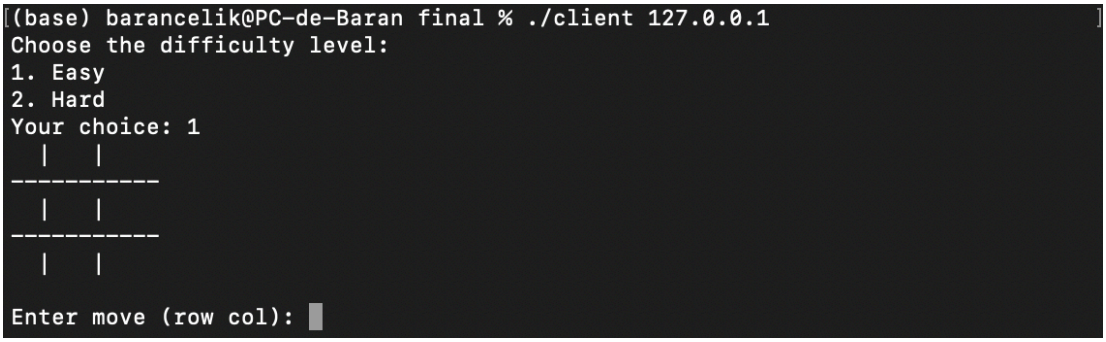

Figure 2 - Entrer le premier coup

Si le joueur entre un coup invalide, le serveur le détecte immédiatement et renvoie un message d'erreur, puis lui redemande un nouveau coup. Un coup est considéré comme invalide dans deux cas : soit les coordonnées saisies sont hors de la grille (exemple 1 4, alors que la grille ne contient que trois colonnes), soit la case ciblée est déjà occupée. Dans les deux situations, le serveur

informe le joueur de l'erreur et lui renvoie la grille actuelle pour qu'il puisse faire un nouveau choix valide.

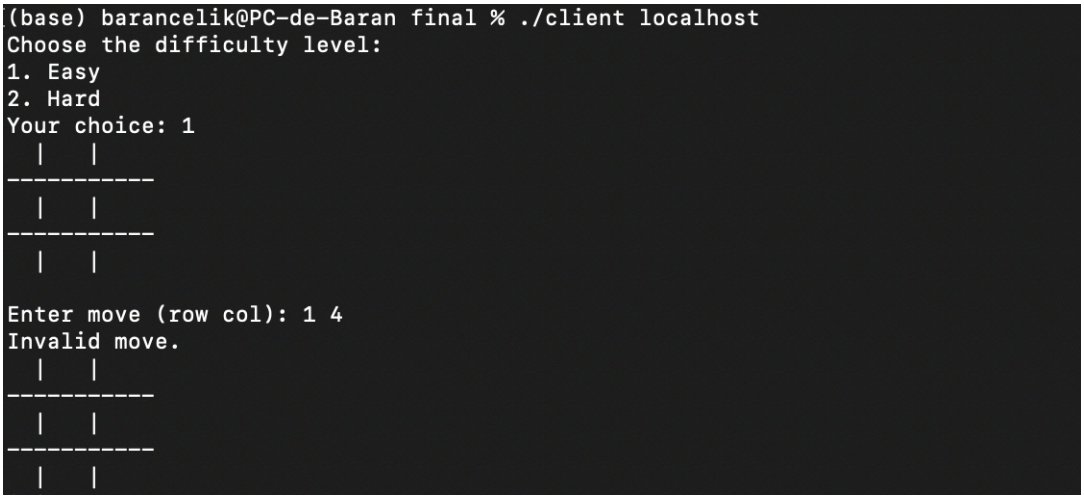

Figure 3 - Coup invalide

Lorsque c'est au tour de l'IA, le serveur génère un coup automatiquement. Si le mode facile est activé, un coup est choisi au hasard parmi les cases libres. En mode difficile, le serveur utilise l'algorithme Minimax pour évaluer tous les coups possibles et sélectionner le plus avantageux. Une fois le coup de l'IA effectué, la grille mise à jour est renvoyée au client pour affichage. Après chaque coup, le serveur vérifie si la partie est terminée à l'aide de la fonction winner(). Cette fonction détermine s'il y a un gagnant, un match nul, ou si la partie doit continuer. En cas de fin de partie, le serveur envoie un message explicite au client : félicitations en cas de victoire, encouragements en cas de défaite, ou indication d'égalité.

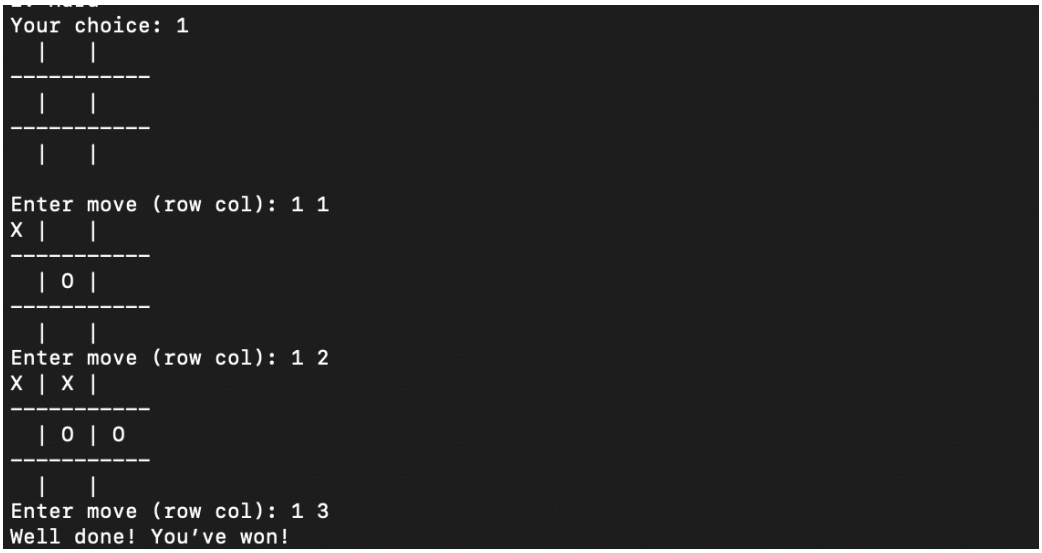

Figure 4- Exemple de victoire

IV. Compilation et lancement

La compilation du projet est facilitée par l'utilisation d'un fichier Makefile, qui automatise la génération des exécutables du client et du serveur. Depuis le répertoire du projet, il suffit d'ouvrir un terminal et d'exécuter la commande **make**. Cette commande compile automatiquement les fichiers source et crée deux exécutables : server pour le serveur, et client pour le client.

Une fois la compilation terminée, on commence par lancer le serveur avec la commande **./server**. Celui-ci se met alors en attente de connexions entrantes. Ensuite, le client peut être lancé avec la commande **./client**, en précisant l'adresse IP du serveur en argument. Par exemple, si le serveur fonctionne sur la même machine, on utilisera simplement **./client localhost**. Une fois la connexion établie, le client peut choisir un mode de jeu et la partie démarre directement dans le terminal, de façon interactive.

V. Conclusion et améliorations possibles

Même si le jeu est déjà fonctionnel, plusieurs idées d'améliorations ont été envisagées pour rendre le jeu plus complet. Tout d'abord, ajouter une *interface graphique* permettrait d'avoir un affichage plus visuel. Dans ce projet, tout se passe dans le terminal, ce qui reste efficace, mais un affichage graphique rendrait le jeu plus accessible et plaisant pour l'utilisateur. Ensuite, il serait utile d'intégrer une possibilité de *rejouer une partie* sans avoir à relancer le client à chaque fois. Après la fin d'une partie, on pourrait tout simplement proposer au joueur de recommencer directement depuis le terminal pour fluidifier le jeu. Une autre idée simple mais intéressante serait de faire *commencer aléatoirement* le joueur ou l'adversaire, afin de ne pas toujours avoir le joueur humain en premier et tester différentes situations de départ. Enfin, une amélioration un peu plus ambitieuse mais très motivante serait d'ajouter un mode PvP. L'idée serait que deux clients puissent se connecter au serveur et jouer l'un contre l'autre à distance.

VI. ANNEXE : Explication de certaines fonctions de rules.h

A. Fonction winner() : détermination de l'issue de la partie

La fonction **winner()** permet de déterminer l'état de la partie à tout moment. Elle est appelée après chaque coup joué pour contrôler si la partie peut se poursuivre ou si elle doit s'arrêter. **winner()** analyse la grille de jeu pour vérifier si un joueur a gagné, si la partie est terminée sur une égalité, ou si elle doit continuer. Son fonctionnement repose sur l'inspection des trois lignes, des trois colonnes et des deux diagonales du plateau. Si l'une de ces combinaisons est entièrement remplie par le même symbole (et qu'elle n'est pas vide), alors le joueur correspondant est déclaré vainqueur, et son symbole (**'X'** ou **'O'**) est retourné. Si aucune victoire n'est détectée mais que la grille est totalement remplie, la fonction retourne **'D'** pour indiquer une égalité. Enfin, si aucune de ces conditions n'est remplie, la fonction retourne le caractère **' '**, ce qui signifie que la partie est toujours en cours.

B. Fonction score() : implémentation de l'algorithme Minimax

Le bot difficile repose sur l'algorithme Minimax, implémenté dans la fonction **score()**. Cet algorithme permet d'anticiper les différents scénarios possibles d'une partie de morpion en évaluant récursivement toutes les configurations futures du plateau de jeu. La fonction score() prend en entrée la grille de jeu **g**, la profondeur de récursion **d**, un booléen indiquant s'il s'agit du joueur maximisant **max**, et le symbole du joueur que l'on souhaite faire gagner **me**. L'adversaire est automatiquement déterminé selon ce symbole.

Elle commence par vérifier si la partie est terminée grâce à la fonction **winner()**. Si c'est le cas, elle attribue un score en fonction du résultat :

- Une victoire du joueur me retourne un score positif, réduit par la profondeur afin de favoriser les victoires rapides : S = 10-d.
- Une défaite retourne un score négatif afin de pénaliser les situations perdantes : S = d-10.
- Une égalité retourne un score nul : S = 0.

Si la partie n'est pas terminée, la fonction visite chaque case vide du plateau. Elle y place temporairement le symbole du joueur courant (le joueur maximisant ou minimisant), puis appelle récursivement **score()** pour évaluer cette nouvelle configuration. Une fois le score obtenu, la case est réinitialisée. L'objectif est ensuite de garder le meilleur score possible :

- Le joueur maximisant cherche à obtenir le score le plus élevé,
- Le joueur minimisant cherche à obtenir le score le plus faible.

La fonction **score()** est utilisée dans **bestMove()**, qui retourne la position du meilleur coup à jouer selon l'évaluation Minimax.

C. Fonction bestMove() : sélection du coup optimal avec Minimax

La fonction **bestMove()** exploite l'algorithme Minimax pour déterminer le meilleur coup à jouer pour un joueur donné dans une situation de jeu donnée. Cette fonction prend en paramètre la grille actuelle du jeu **g** et le symbole du joueur **me**, soit **'X'** ou **'O'**. Son objectif est d'explorer tous les coups possibles (cases encore vides) et, pour chacun, d'estimer la valeur associée à ce coup en appelant la fonction **score()**.

Pour ce faire, la fonction parcourt chaque case de la grille. Lorsqu'elle rencontre une case vide, elle y place temporairement le symbole du joueur **me** et appelle **score()** en simulant le tour suivant de l'adversaire. Le score obtenu représente l'efficacité du coup testé, en tenant compte de tous les scénarios futurs. La fonction conserve en mémoire le meilleur score rencontré ainsi que la position associée. Une fois tous les coups possibles évalués, elle retourne les coordonnées **{ligne, colonne}** du coup optimal.

D. Fonction randomMove() : sélection aléatoire d'un coup

La fonction **randomMove()** constitue l'implémentation d'un mode facile pour le jeu de morpion. Contrairement à la fonction **bestMove()** qui utilise l'algorithme Minimax pour prendre des

décisions optimales, **randomMove()** se contente de choisir un coup valide de manière totalement aléatoire. Elle prend en paramètre la grille de jeu actuelle **g** et retourne une paire de coordonnées **{ligne, colonne}** correspondant à une case vide sélectionnée au hasard.

Le fonctionnement de la fonction est simple :

- Elle parcourt toute la grille pour collecter dans un vecteur **moves** l'ensemble des positions encore disponibles (cases contenant le symbole vide **' '**).
- Si aucune case vide n'est trouvée, elle retourne **{-1, -1}** pour indiquer qu'aucun coup n'est possible.
- Sinon, elle utilise la fonction **rand()** pour tirer au hasard un indice dans la liste des coups disponibles, puis retourne la position correspondante.