Rapport : Création d'un jeu de Puissance 4

Par Cassius Garat & Simon Hourlier

Quelques mots pour débuter :

Compilation : gnatmake beat_cumputer.adb

Le but de ce projet est de créer un jeu de Puissance 4 où l'utilisateur dispose d'un affichage graphique simple et le choix de jouer contre l'ordinateur ou un adversaire humain. Pour créer ce jeu, nous avons d'abord instancié une forme générique pouvant s'adapter à tous jeux opposant 2 joueurs et un équilibre gagnant/perdant (Un coup favorable pour un joueur et autant défavorable pour son adversaire). Puis, nous avons adapté cette généricité au jeu de Puissance 4. Dans ce rapport, nous faisons le choix d'expliquer deux fichiers : Puissance4.adb et Moteur_Jeu.adb car ce sont les plus intéressants pour le fonctionnement du jeu.

I/ Définition des termes clés

La première étape était de coder le package Puissance4, et de tester une partie entre deux humains. Dans ce fichier sont définies les bases de ce projet.

Le package Puissance4 définit un type Coup, un type Joueur et un type Etat, et a trois paramètres génériques : la largeur du plateau de jeu, sa hauteur, et l'alignement de pions nécessaire pour gagner.

- Un Joueur est une chaîne de caractère (son nom), et un symbole.
- Un Coup est alors un joueur, et un numéro de colonne.
- Un Etat est un plateau de jeu représenté par une matrice de taille Hauteur*Largeur, et deux joueurs : Joueur1 et Joueur2.

II/ Partie entre deux humains

Il est évidemment plus facile de coder la partie entre deux humains plutôt que d'affronter un ordinateur car tout le processus décisionnel n'est pas implémenté.

La première étape consiste à coder une fonction pour initialiser l'état, qui remplit le plateau de symboles ' ' et initialise le symbole de Joueur1 à 'X', et celui de Joueur2 à 'O'. Ainsi, nous obtenons une grille qui se remplira par des X ou des O.

Pour déterminer l'évolution du jeu après un Coup c'est à dire l'état suivant, il suffit de parcourir de bas en haut la colonne renseignée par le Coup en paramètre et de trouver la première case non vide. On y insère ensuite le symbole correspond au joueur ayant réalisé ce coup.

La première fonction qui coûte cher est celle qui détermine si un joueur est gagnant.

Il suffit de compter les pions dans chaque ligne, colonne et diagonale, et de déterminer s'il y a un alignement correspondant à l'alignement générique.

Pour que cela soit plus pratique, il était plus commode de décomposer cette fonction en quatre parties, l'une déterminant si une ligne comporte un alignement, si une colonne comporte un alignement et enfin si une des diagonales en comporte un.

Dans ces fonctions, certaines cases (en fait, un grand nombre), sont parcourues plusieurs fois. Le coût est alors plus élevé que O(Hauteur*Largeur), même en activant un booléen lorsqu'on trouve un alignement, afin de quitter la fonction directement, puisque dans le pire cas (lorsque le joueur n'est pas gagnant), on parcourt quand même tout le plateau.
La meilleure idée pour améliorer l'algorithme serait alors de marquer certaines cases pour ne plus les parcourir.

Pour déterminer si un état est nul, on compte le nombre de cases pleines et on compare le résultat à Largeur*Hauteur, ce qui donne un coût en O(Hauteur*Largeur).

Les dernières fonctions/procédures utiles pour commencer une partie entre humains sont le choix du coup d'un joueur en lui demandant via le terminal, et l'affichage des coups.

Le principal étant fait, il ne reste plus qu'à coder le package Partie.
Ce package contient la procédure "Jouer_Partie", qui se déroule de la manière suivante :

Tant que le joueur appelant n'est pas gagnant, que son adversaire ne l'est pas non plus et qu'il n'y a pas nul, alors on lui demande ce qu'il veut jouer, on affiche le coup et le jeu. Si la situation n'est pas terminale (match nul ou victoire du joueur appelant), on demande ensuite à son adversaire de jouer.

Résultats :

Les parties entre deux humains se déroulent bien, il n'y a pas de latence puisque les algorithmes utilisés sont basiques pour l'instant.

III/ Partie contre un ordinateur

Jouer contre un ami est sympathique, mais la machine n'apporte pas grand-chose comparée à une feuille de papier. (Même pas l'aspect graphique…)

Nous arrivons donc à la partie intéressante. Après avoir géré les parties basiques du jeu, nous avons codé Moteur_Jeu.adb. Ce fichier contient les mécanismes décisionnels de l'ordinateur afin qu'il puisse jouer contre un humain, et de façon réfléchie.

Pour cela, il faut savoir si une situation est favorable ou pas, et quels sont les éléments qui la rendent justement favorable.

a) Evaluation statique d'un état

La première étape fut donc d'implémenter un algorithme pour l'évaluation statique d'un état.

Pour un puissance 4, un joueur est d'autant plus confortable dans sa position qu'il a d'alignements "utiles", c'est à dire d'alignements qui sont adjacents à une ou plusieurs cases vides dans la direction de l'alignement.
On ajoute alors un certain poids si on trouve un alignement suivi d'une case vide, et un poids encore plus important si l'alignement est suivi de deux cases vides (on se place dans le cas du puissance4 classique, c'est à dire (6,7,4), donc on se limite à chercher au maximum deux cases vides).

Le programme en résultant est alors assez semblable au programme permettant de savoir si un joueur est gagnant. On décompose le programme en quatre parties, en ajoutant un poids selon la ligne choisie, la colonne choisie ou sur les diagonales.
Ensuite le programme global utilise ces trois sous-programmes appliqués au joueur visé et à son adversaire, et la différence entre son score et celui de son adversaire donne l'évaluation statique.

Le coût est grosso-modo en O(Largeur*Hauteur), à quelques multiples près.

Il est sans aucun doute possible d'améliorer l'évaluation statique du puissance4, mais cela reste un jeu assez basique, et compte tenu des résultats déjà très satisfaisants avec cette fonction d'évaluation, cela aurait été de la pure curiosité.

b) Algorithme Min-max

La seconde partie intéressante du moteur de jeu est cet algorithme.
La fonction Eval_Min_Max permet de parcourir un arbre dont les noeuds sont les sont des coups simulés à partir d'un certain état.
La fonction Choix_Coup utilise la fonction précédente, et enregistre le coup qui a la meilleure évaluation, c'est à dire qui a le moins de bonnes issues pour l'adversaire, et le plus de bonnes issues pour le joueur.

La complexité de Eval_Min_Max est dans le pire des cas en O(a^n), où a est le nombre de coups possibles maximum, et n la profondeur. Ce cas correspond au cas où les colonnes se remplissent progressivement, sans qu'une colonne se remplisse plus rapidement que les autres.
L'algorithme construit un arbre, récursivement, avec autant de noeuds que de coups possibles à chaque étape. On arrive donc à O(a^n) au maximum, mais en réalité la complexité est un peu moins importante, puisqu'il n'y a pas toujours a coups possibles.

Le résultat est assez bluffant car il est difficile de battre l'ordinateur à partir d'une profondeur de 5. Le temps de calcul est d'un peu moins de 3 secondes pour cette profondeur, pour chaque coup joué par l'ordinateur. Si l'on ne souhaite pas attendre ces 3 secondes il est possible de baisser la profondeur mais nous avons trouvé que c'était un bon compromis Performance-Rapidité.

Résultats :

L’algorithme permet de jouer contre un ordinateur performant. L’ordinateur est difficile à battre et il ne réfléchit pas trop longtemps.

Bonus :

Nous avons également testé de faire jouer deux ordinateurs ensemble. Il est assez intéressant de découvrir que ce n'est pas toujours le même joueur qui gagne. Cette option peut devenir plus qu'intéressante en développant du "machine learning" en faisant jouer un grand nombre de fois la machine contre elle-même. Nous pourrions ainsi former l'ordinateur à jouer comme un professionnel qui nous humilierait par la suite.

IV / Prise en main

Comme indiqué précédemment, il faut d'abord compiler beat_computer.adb puis l'exécuter. La grille de départ d'affiche et nous pouvons ensuite nous laisser guider.
Il y'a 3 choix qui s'offrent à nous :

1) Jouer contre l'ordinateur
2) Jouer contre un ami
3) Regarder l'ordinateur jouer tout seul (Il n'y a pas de honte à cela)

Nous aurions pu rajouter une requête pour le nom des joueurs mais nous n'avons pas pensé cela indispensable. Il faut donc faire avec le champion du monde d'échecs Carlsen contre son récent challenger Karjakin.
Ensuite place au jeu et bonne chance si vous avez choisi la première option ! (Karjakin est coriace)

Le petit mot de la fin :

Etant tout deux passionné d'échecs, nous avons l’ambition de développer un jeu d’échecs pendant les vacances en récupérant la partie générique. L’évaluation statique est assez difficile pour les échecs, même pour des grands-maîtres. On peut néanmoins tenter d’évaluer une situation selon la valeur des pièces dans chaque camp, et puis en comptant le nombre de pièces qui attaquent une pièce donnée et le nombre de défenseurs. On peut également évaluer une position selon la position du roi, etc …
Le problème majeur est plutôt le nombre impressionnant de coups possibles. Il faudrait impérativement passer l’algorithme en version Alpha Bêta.

Ce projet nous a permis de découvrir la généricité et des algorithmes intéressants tels que le Min-max autour d'une conception d'un jeu simple. Nous nous sommes ainsi bien familiariser avec le langage ADA et nous pouvons désormais nous essayer à des jeux plus complexes si le temps et la motivation nous le permet.