TP numéro 5
23 et 26 novembre 1999

Programmation en binôme : les algorithmes $MiniMax$ et $\alpha - \beta$

1 Description des algorithmes

Définition d'un jeu à deux joueurs

Un *jeu à deux joueurs* est défini classiquement comme un arbre qui a comme nœuds des *positions* (la racine est appelée la «position initiale du jeu»). Chaque nœud est un nœud «joueur» (i.e. c'est au joueur de jouer son coup) ou un nœud «opposant», et les joueurs s'alternent, de telle sorte que un nœud joueur a comme fils une liste de nœuds opposants, que l'on obtient à partir des différents coups disponibles (symétriquement pour opposant). Si un nœud n'a pas de fils, c'est un nœud terminal, et dans ce cas on associe à ce nœud à l'aide d'une fonction h : *nœudterminal* $\rightarrow \mathbb{R} \cup \{\pm\infty\}$ une *valeur* qui indique ce que le joueur a gagné ou perdu. Typiquement, si on s'intéresse seulement à des jeux comme les échecs, on aura $+\infty$ pour gain, $-\infty$ pour perte et 0 pour les autres positions (pat et autres nuls), mais il y a des jeux avec des structures d'évaluation plus complexes, par exemple les jeux où on mise de l'argent.

D'une certaine façon, cet arbre contient toutes les parties possibles que l'on peut jouer à partir de la position initiale. Si on a la possibilité d'explorer tout l'arbre, on peut déterminer s'il existe une stratégie gagnante pour le joueur. Si l'arbre est trop gros (ou si l'on impose des limites de temps) on se contente normalement d'explorer le sous-arbre obtenu en tronquant l'arbre de jeu à une certaine profondeur, et en évaluant les nœuds tronqués à l'aide d'une heuristique h'.

Dans les deux cas, il est déraisonnable de supposer que l'arbre de jeu est donné de façon extensive, et on préfère le présenter de façon implicite par des «règles de jeu» qui disent comment obtenir la liste des fils d'une position donnée.

La visite $MiniMax$

L'algorithme $MiniMax$, dû à Von Neumann, est très simple : on visite l'arbre de jeu pour faire remonter à la racine une valeur (appelée «valeur du jeu») qui est calculée récursivement de la façon suivante :

- $MiniMax(p) = h(p)$ si p est une position terminale
- $MiniMax(p) = \max(MiniMax(o_1), ..., MiniMax(o_n))$ si p est une position joueur avec fils $o_1, ..., o_n$
- $MiniMax(p) = \min(MiniMax(j_1), ..., MiniMax(j_m))$ si p est une position opposant avec fils $j_1, ..., j_m$

On peut vérifier que $MiniMax(p)$ est la meilleure valeur possible à partir de la position p, si l'opposant joue de façon optimale. Il est clair que pour appliquer l'algorithme MiniMax il suffit de disposer de

- un type de données *position* qui permet de représenter les états possibles du jeu (et une position initiale) ;
- une fonction h : *position* $\rightarrow \mathbb{R} \cup \{\pm\infty\}$ pour évaluer les positions terminales ;
- une fonction $estjoueur$: *position* $\rightarrow \{oui, non\}$ qui sert à déterminer si une position est une position joueur ou opposant ;

– une fonction *accessibles* qui prenne une position p et retourne la liste des positions accessibles par les coups disponibles à partir de p.

Cela nous donne notre interface C :

```
/* les positions */
typedef struct {
  . . . /* Ceci dépend du jeu
         * et ne doit pas être utilisé par la visite de l'arbre */
} position;

/* une liste de positions */
typedef struct {
  int nombre;
  position tableau[];
  /* On utilise un tableau, dont le nombre d'éléments est ci-dessus */
  /* Les coups à jouer sont représentés par leur position dans le tableau */
} posliste;

/* enfin, les fonctions propres au jeu */
double h(position p);
int estjoueur(position p);
posliste accessibles(position p);
```

La visite $\alpha - \beta$

L'algorithme $\alpha-\beta$ est une optimisation du MiniMax, qui «coupe» des sousarbres dès que leur valeur devient inintéressante aux fins du calcul de la valeur MiniMax du jeu. On s'intéressera donc, sur chaque nœud, en plus de la valeur, à deux autres quantités, nommées α et β, qui seront utilisées pour calculer la valeur du nœud.

α d'un nœud est une approximation par le bas de la vraie valeur du nœud. Elle est égale à la valeur sur les feuilles, et est initialisée à $-\infty$ ailleurs. Ensuite, sur les nœuds joueur elle est maintenue égale à la plus grande valeur obtenue sur les fils visités jusque là, et elle est égale à la valeur α de son prédécesseur sur les nœuds opposant.

β d'un nœud est une approximation par le haut de la vraie valeur du nœud. Elle est égale à la valeur sur les feuilles, et est initialisée à $+\infty$ ailleurs. Ensuite, sur les nœuds opposant elle est maintenue égale à la plus petite valeur obtenue sur les fils visités jusque là, et elle est égale à la valeur β de son prédécesseur sur les nœuds joueur.

L'algorithme ALPHA-BETA peut être décrit par le pseudo-code suivant :

```
function ALPHA-BETA(P, A, B) /* ici A est toujours inférieur à B */
begin
  si P est une feuille alors
    retourner la valeur de P
  sinon
    initialiser Alpha de P à -∞ et Beta de P à +∞;
    si P est un nœud Min alors
      pour tout fils Pi de P faire
        Val = ALPHA-BETA(Pi, A, Min(B,Beta of P))
        Beta de P = Min(Beta de P, Val)
```

```
      Si A >= Beta de P /*ceci est la coupure alpha */
      alors returner Beta de P
    finfaire
    returner Beta de P
  sinon
    pour tout fils Pi de P faire
      Val = ALPHA-BETA(Pi, Max(A,Alpha de P), B)
      Alpha de P = Max(Alpha de P, Val)
      Si Alpha de P >= B /*ceci est la coupure beta */
      alors returner Alpha de P
    finfaire
    returner Alpha de P;
end ALPHA-BETA;
```

On sait que la véritable valeur MiniMax v d'un nœud est encadrée par α et β (i.e. $\alpha \leq v \leq \beta$), et si on appelle la fonction ALPHA-BETA avec les valeurs $(P, -\infty, +\infty)$ on obtient précisément MiniMax(P). ALPHA-BETA permets assez souvent de doubler la profondeur d'exploration d'un arbre à parité de ressources, par rapport à MiniMax.

2 Travaux pratiques

Programmer à plusieurs

Si vous avez à gérer un projet assez important, il est indispensable de pouvoir le décomposer en sous-projets dont les interfaces sont clairement spécifiées, pour permettre que les différentes composantes puissent être developpées en parallèle tout en garantissant la cohérence du résultat final.

Nous allons faire un essai à petite échelle dans le cadre de ce TP : le but est de programmer des algorithmes classiques de visite d'arbres de jeu, d'une part, et de programmer un jeu particulier d'autre part. Ces deux modules, ensemble, nous donneront un «joueur» automatique pour le jeu voulu. Pour cela, regroupez vous en binômes (programmeur A, programmeur B).

- Tâches du programmeur A :
  1. Programmation de MiniMax
  2. Programmation du jeu Puissance 4
- Tâches du programmeur B :
  1. Programmation du jeu Tic-Tac-Toe
  2. Programmation de AlphaBeta

En pratique, chaque programmeur écrit les fonctions dont il a la charge dans un fichier C (par exemple `visite.c` et `regles.c`). Ces fichiers incluent le fichier «header» (par exemple `jeu.h`, avec la ligne `#include "jeu.h"`) qui définit l'interface commune. Un troisième fichier C (`jeu.c`) sert à tester ces fonctions, et on compile par exemple avec `gcc -g -O2 -Wall -o jeu visite.c regles.c jeu.c`.

Exercice 1 (MiniMax et Tic-Tac-Toe) . *Le programmeur A écrit les fonctions de l'algorithme MiniMax. Le programmeur B écrit l'interface du jeu Tic-Tac-Toe. Compilez ensemble vos deux modules et testez leur correction de deux façons :*
en faisant un programme qui calcule la valeur du jeu Tic-Tac-Toe,
en faisant un programme qui permette de jouer à Tic-Tac-Toe contre l'ordinateur.

Exercice 2 (AlphaBeta et Puissance 4) . *Le programmeur A écrit l'interface du jeu Puissance 4. Le programmeur B écrit les fonctions de l'algorithme AlphaBeta. Compilez ensemble ces deux modules.*
Essayez aussi la combinaison MiniMax + Puissance 4 et la combinaison AlphaBeta + Tic-Tac-Toe.

Pour ceux qui sont rapides

Exercice 3 *Adapter les exercices ci-dessus aux jeux à plus de deux joueurs. Application : le jeux des dames chinoises.*

2.1 Visite de l'arbre de jeu

Pour chaque type de visite, on écrit deux fonctions, en utilisant exclusivement les fonctions de l'interface :
une fonction `double val(position p)` calcule la valeur du jeu,
une fonction `int val(position p)` dit quel est le meilleur coup pour le joueur qui doit jouer (on retourne sa position dans la `posliste`).

Ensuite, on modifie ces fonctions pour permettre de spécifier la profondeur maximale d'exploration de l'arbre de jeu.

2.2 Interface de chaque jeu

Pour chaque jeu, il faut définir le type de données `position` et programmer les fonctions suivantes :

```
/* affiche le jeu dans la postition p */
void affiche(position p);

/* donne la position de départ */
position start();

/* Fonctions de l'interface */
double h(position p);
int estjoueur(position p);
posliste accessibles(position p);
```

Rappels : règles des jeux

Tic-Tac-Toe

sur une grille 3×3, chaque joueur coche à tour de rôle une case, celui qui en aligne 3 a gagné.

Puissance 4

sur une grille de 7 colonnes de 6 cases, chaque joueur met à tour de rôle un pion dans une colonne, celui-ci descend en bas de la colonne. Celui qui en aligne 4 a gagné.

Dames chinoises

Sur les intersections d'un damier en étoile, chacun des six joueurs a 10 pions de sa couleur, chacun dans l'une des pointes de l'étoile. Le but du jeu est de tous les déplacer dans la pointe opposée. Chacun à son tour déplace un pion, de l'une des deux façons suivantes : vers une case libre contiguë, ou bien en faisant un nombre quelconque de sauts avec ce pion. Chaque saut se fait vers une case libre, par dessus une case occupée.