MMFAI
Cours de programmation

TP numéro 6
semaine du 4 janvier 1999

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

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).

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 joueur 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 $h : \textit{nœudterminal} \to R \cup \{\pm\infty\}$ 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 MiniMax (appelee "valeur du jeu") qui est calculée récursivement de la façon suivante :
- $MiniMax(p) = h(p)$ si p est une position terminale
- $MiniMax(p_j) = max(MiniMax(o_1), \ldots, MiniMax(o_n))$ si p_j est une position joueur avec fils $o_1, \ldots, o_n$
- $MiniMax(p_o) = min(MiniMax(p_1), \ldots, MiniMax(p_m))$ si p_o est une position joueur avec fils $p_1, \ldots, p_m$

Il est clair que pour appliquer l'algorithme MiniMax il suffit de disposer de
– un type de donnée "position" (et une position initiale "start")
– un type de donnée "coup"
– une fonction $h : position \rightarrow double$ pour évaluer les positions terminales
– une fonction $estjoueur : position \rightarrow int$ qui sert à déterminer si une position est une position joueur ou opposant
– une fonction $fils\_de$ qui prenne une position p et retourne une liste de couples $(pacc, coup)$, des positions accessibles par les coups disponibles sur p

Cela nous donne notre interface C :

```
/* les positions */
typedef struct {
. . .
} position;

/* les coups */
typedef struct {
. . .
} coup;

/* une couple position, coup */
typedef struct {position fst;
                coup snd;} pc;

/* avec les fontions: fts et snd */
position fst(pc c);
coup snd(pc c);

typedef struct {
  . . .
} pclist;

/* plus, ici, les fonctions habituelles sur les listes:
   hd, tail, nil et isempty */

/* enfin, les fonctions propres au jeu */

double h(position p);

int estjoueur(position p);

pclist fils_de(position p);
```

On rajoutera aussi une fonction qui imprime une position, pour visualiser les résultats.

```
void imprime(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
        returner 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.

1.1 Tâches du programmeur A

Exercice 1.1 (Programmation de MiniMax) *Écrivez une fonction* `int val(position p)` *qui calcule la valeur d'un jeu avec MiniMax, et retourne une partie (liste de positions,coups) réalisant cette valeur, en utilisant exclusivement les fonctions dans l'interface.*

Exercice 1.2 (Programmation de AlphaBeta) *Écrivez une fonction* `int valab(position p)` *qui calcule la valeur d'un jeu avec l'agorithme* $\alpha - \beta$*, et retourne une partie (liste de positions,coups) réalisant cette valeur, en utilisant exclusivement les fonctions dans l'interface.*

Exercice 1.3 (Profondeur fixe) *Modifiez vos algorithmes pour permettre de spécifier la profondeur maximale d'exploration de l'arbre de jeu.*

1.2 Tâches du programmeur B

Exercice 1.4 (Programmation du jeu Tic-Tac-Toe) *Définissez les types de données et programmez les fonctions dans l'interface définie plus en haut pour le jeu du Tic-Tac-Toe. (Rappel : sur une grille* 3×3*, chaque joueur coche à tour de rôle une case, celui qui en aligne 3 a gagné.)*

Exercice 1.5 (Programmation du jeu Puissance 4) *Définissez les types de données et programmez les fonctions dans l'interface définie plus en haut pour le jeu Puissance 4. (Rappel : 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.)*

1.3 Tâches communes à A et B

Exercice 1.6 *Compilez ensembe vos deux modules et ... jouez.*

N'oubliez pas de vous déconnecter avant d'abandonner le clavier (voir notes du stage Unix).