

L'algorithme min-max

Par martin9_36



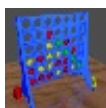
OPENCLASSROOMS

www.openclassrooms.com

*Licence Creative Commons 7 2.0
Dernière mise à jour le 14/04/2011*

Sommaire


Sommaire	2
L'algorithme min-max	3
Principe	3
Pour résumer	4
L'algorithme	4
Fonction Min	5
Fonction Max	6
Fonction eval	6
TP : IA pour morpion	7
Partager	13




L'algorithme min-max

Par  martin9_36


Mise à jour : 14/04/2011

Difficulté : Intermédiaire 



Ne vous-êtes vous jamais demandé, en jouant au échecs ou encore à reversi sur ordinateur, "Mais comment fait-il pour réfléchir ?" 

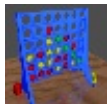
Ce tutoriel est là pour vous apporter une ébauche de réponse en étudiant un des plus simples algorithmes cognitifs : l'algorithme min-max. Malgré sa simplicité, cet algorithme peut s'appliquer de manière relativement efficace à bon nombre de jeux dont les deux joueurs jouent à tour de rôle.

Cependant, sa portée ne s'étend pas au-delà de ce type de jeux. D'autres algorithmes plus compliqués sont nécessaires dans ce genre de cas. Il constitue cependant une bonne approche au monde de l'intelligence artificielle. 

Pour se faire comprendre d'un maximum de programmeurs, aucun langage n'est utilisé dans ce tutoriel, sauf pour le TP, dont le code en C est facilement transposable en un autre langage.

C'est parti !

Sommaire du tutoriel :



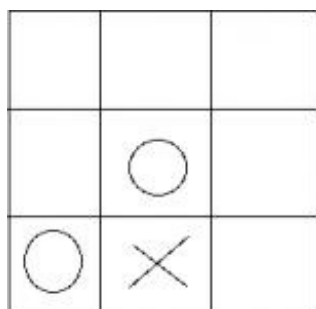
- [Principe](#)
- [L'algorithme](#)
- [TP : IA pour morpion](#)

Principe

L'algorithme min-max est particulièrement adapté aux jeux se jouant à tour de rôle, comme par exemple le morpion ou le puissance 4, parmi les plus simples.

Le but de min-max est de trouver le meilleur coup à jouer à partir d'un état donné du jeu.

Exemple avec un jeu de morpion :



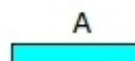
Ici, le meilleur coup à jouer pour le joueur des croix est la case en haut à droite



Et comment je fais ça moi ? 

Le principe consiste à construire un arbre de jeu : c'est un arbre qui représente toutes les évolutions possibles du jeu à partir de la configuration actuelle.

La racine (le nœud A) représente l'état actuel du jeu, et la question est ici de savoir si le meilleur coup consiste en B1



ou B2.

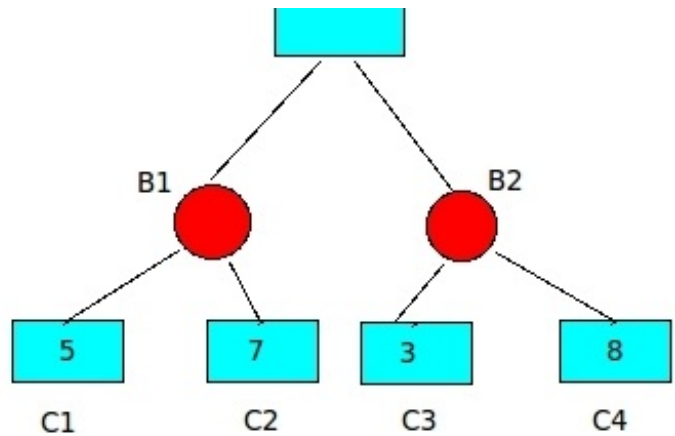
On va donc se demander ce que va jouer l'adversaire si l'on joue B1, puis B2, et choisir le coup qui nous avantagera le plus, c'est-à-dire donner un "poids" à B1 et B2.

En effet, si on joue B1, l'adversaire va choisir le coup qui nous avantage le moins entre C1 et C2, c'est-à-dire C1, qui a le poids le plus faible.

Donc si on joue B1, on arrivera à la configuration C1, dont le poids est 5.

De même, si on joue B2, l'adversaire nous mènera en C3, dont le poids est 3.

On peut donc dire que le poids de B1 est 5, et celui de B2, 3. On choisira donc B1, et on recommence à partir de C1.



On remarque que l'algorithme suppose que l'adversaire jouera toujours le meilleur coup possible

Pour résumer

On va choisir le maximum des nœuds fils de A, sachant que chaque nœud fils de A est lui-même le minimum de ses nœuds fils, et que chaque nœud fils des nœuds fils de A est lui-même le maximum des ses propres nœuds fils, etc... jusqu'à arriver aux feuilles qui symbolisent la fin de la partie.

Pour un jeu de morpion, parcourir tout l'arbre à chaque fois n'est pas gênant, mais par contre, pour des jeux plus complexes tels que les échecs, on peut atteindre les 10^{13} combinaisons, ce qui représente beaucoup d'attente avant chaque coup. 🤔

On restreint donc le parcours de l'arbre à une certaine profondeur n. L'algorithme min-max est donc caractérisé par sa profondeur.

On applique donc un processus qu'on répétera jusqu'à atteindre les feuilles ou la profondeur voulue.



D'accord, mais comment donner un poids aux feuilles ? 🤔

Effectivement, quand on arrive aux feuilles, on n'a plus de nœuds fils dont tirer un poids... Il faut donc arriver à évaluer une configuration du jeu, à lui donner un poids.

C'est justement ce qui différencie deux algorithmes min-max : la fonction d'évaluation.

Il est possible d'isoler deux cas :

- Le jeu est fini : si on a gagné, on associe un poids élevé, et si c'est l'adversaire qui a gagné, on associe un poids très négatif, et 0 si personne ne gagne
- Le jeu n'est pas fini : à ce moment-là, il faut trouver un moyen de quantifier l'avantage que nous procure la configuration du jeu.

On se penchera sur la question dans la partie suivante 🤔.

L'algorithme

Une des idées fondamentales de min-max est de simuler les coups, pour analyser leur impact sur le déroulement de la partie. On aboutit donc aisément à l'algorithme suivant, dont nous allons détailler ensemble les fonctions utilisées :

Code : Autre

```

fonction jouer : void

    max_val <- -infini

    Pour tous les coups possibles
        simuler(coup_actuel)
        val <- Min(etat_du_jeu, profondeur)

        si val > max_val alors
            max_val <- val
            meilleur_coup <- coup_actuel
        fin si

    annuler_coup(coup_actuel)
fin pour

jouer(meilleur_coup)
fin fonction

```



Un problème qui peut se poser dès à présent est : comment simuler les coups ?

En effet, s'il est assez facile de jouer, puis d'annuler un coup dans un jeu comme le morpion, l'annulation devient difficile si on parle de reversi par exemple. 😞

Dans les cas où il est délicat, voire impossible d'annuler un coup, il est envisageable de créer une copie du plateau de jeu avant de simuler un coup, puis de le restaurer ensuite à l'aide de la copie.

Fonction Min

Le but de la fonction Min est de trouver le minimum des nœuds-fils du nœud envoyé en paramètre.
Or chaque nœud-fils est le maximum de ses propres nœuds-fils



Ne pas oublier de s'arrêter au bon moment ! 😋

On aboutit donc à :

Code : Autre

```

fonction Min : entier

    si profondeur = 0 OU fin du jeu alors
        renvoyer eval(etat_du_jeu)

    min_val <- infini

    Pour tous les coups possibles
        simuler(coup_actuel)
        val <- Max(etat_du_jeu, profondeur-1)

        si val < min_val alors
            min_val <- val
        fin si

    annuler_coup(coup_actuel)
fin pour

renvoyer min_val
fin fonction

```

Bien sûr, on envoie profondeur-1 en paramètre à Max, et non pas profondeur, sinon on parcourt l'arbre tout entier. 😊

Fonction Max

Elle est en tout point semblable à la fonction Min, sauf qu'au lieu de chercher le minimum de ses nœuds-fils, elle en cherche le maximum. 😊

On a donc :

Code : Autre

```
fonction Max : entier
    si profondeur = 0 OU fin du jeu alors
        renvoyer eval(etat_du_jeu)

    max_val <- -infini

    Pour tous les coups possibles
        simuler(coup_actuel)
        val <- Min(etat_du_jeu, profondeur-1)

        si val > max_val alors
            max_val <- val
        fin si

        annuler_coup(coup_actuel)
    fin pour

    renvoyer max_val
fin fonction
```

Fonction eval

Contrairement aux autres fonctions, la fonction d'évaluation dépend presque entièrement du jeu. 😊



Une bonne fonction d'évaluation peut faire une grande différence entre deux algorithmes min-max

Tentons de trouver une fonction d'évaluation pour le morpion

- Si la partie est finie :

Si on a gagné, on renvoie un grand nombre (1000 par exemple) - le nombre de coups (on va donc essayer de gagner le plus vite possible).

Si on a perdu, on renvoie un nombre très négatif (-1000 par exemple) + le nombre de coups (pour essayer de survivre le plus longtemps possible 😊).

Et si personne n'a gagné, on renvoie 0.

- Si la partie n'est pas finie :

On renvoie le nombre de séries de deux pions alignés du joueur, moins celui de l'adversaire.



Il y a mieux comme fonction d'évaluation, mais ce n'est qu'un exemple. 🤖

On a donc l'algorithme :

Code : Autre

```
fonction eval : entier
    si le partie est finie alors
        si joueur a gagné alors
            renvoyer 1000-nombre_de_coups
        sinon si adversaire a gagné alors
            renvoyer -1000+nombre_de_coups
        sinon
            renvoyer 0
    fin si
    renvoyer nb_de_series_de_2_joueur -
    nb_de_series_de_2_adversaire
fin fonction
```

Voilà, on est arrivé à la fin de cette partie.

Place au TP 🧑🎓

TP : IA pour morpion

On modélise le plateau de jeu par un tableau d'entiers 3*3.

Si $\text{jeu}[i][j] = 0$, la case est libre, si $\text{jeu}[i][j] = 1$, la case est occupée par le joueur 1, et si $\text{jeu}[i][j] = 2$, la case est occupée par le joueur 2.

Écrivez les fonctions suivantes dont on donne le prototype :

Code : C

```
void IA_jouer(int** jeu, int profondeur);
int Max(int** jeu, int profondeur);
int Min(int** jeu, int profondeur);
int eval(int** jeu);
int gagnant(int** jeu);
```

La correction sera donnée en langage C, mais la transposition en un autre langage est tout à fait réalisable. 😊

C'est parti! 🧐

Solution :

Secret (cliquez pour afficher)

IA_jouer:

Code : C

```
void IA_jouer(int** jeu,int profondeur)
{
    int max = -10000;
    int tmp,maxi,maxj;
    int i,j;

    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            if(jeu[i][j] == 0)
            {
                jeu[i][j] = 1;
                tmp = Min(jeu,profondeur-1);

                if(tmp > max)
                {
                    max = tmp;
                    maxi = i;
                    maxj = j;
                }
                jeu[i][j] = 0;
            }
        }

        jeu[maxi][maxj] = 1;
    }
}
```

Max:

Code : C

```
int Max(int** jeu,int profondeur)
{
    if(profondeur == 0 || gagnant(jeu)!=0)
    {
        return eval(jeu);
    }

    int max = -10000;
    int i,j,tmp;

    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            if(jeu[i][j] == 0)
            {
                jeu[i][j] = 2;
                tmp = Min(jeu,profondeur-1);

                if(tmp > max)
                {
                    max = tmp;
                }
                jeu[i][j] = 0;
            }
        }
    }

    return max;
}
```


}

Min :

Code : C

```

int Min(int** jeu, int profondeur)
{
    if(profondeur == 0 || gagnant(jeu)!=0)
    {
        return eval(jeu);
    }

    int min = 10000;
    int i, j, tmp;

    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
        {
            if(jeu[i][j] == 0)
            {
                jeu[i][j] = 1;
                tmp = Max(jeu, profondeur-1);

                if(tmp < min)
                {
                    min = tmp;
                }
                jeu[i][j] = 0;
            }
        }
    }

    return min;
}

```

Eval :

Code : C

```

void nb_series(int** jeu, int* series_j1, int* series_j2, int n)
//Compte le nombre de séries de n pions alignés de chacun des
//joueurs
{
    int compteur1, compteur2, i, j;

    *series_j1 = 0;
    *series_j2 = 0;

    compteur1 = 0;
    compteur2 = 0;

    //Diagonale descendante
    for(i=0; i<3; i++)
    {
        if(jeu[i][i] == 1)
        {
            compteur1++;
            compteur2 = 0;

            if(compteur1 == n)
            {

```

```
        *series_j1++;
    }
}
else if(jeu[i][i] == 2)
{
    compteur2++;
    compteur1 = 0;

    if(compteur2 == n)
    {
        *series_j2++;
    }
}

compteur1 = 0;
compteur2 = 0;

//Diagonale montante
for(i=0;i<3;i++)
{
    if(jeu[i][2-i] == 1)
    {
        compteur1++;
        compteur2 = 0;

        if(compteur1 == n)
        {
            *series_j1++;
        }
    }
    else if(jeu[i][2-i] == 2)
    {
        compteur2++;
        compteur1 = 0;

        if(compteur2 == n)
        {
            *series_j2++;
        }
    }
}

//En ligne
for(i=0;i<3;i++)
{
    compteur1 = 0;
    compteur2 = 0;

    //Horizontalement
    for(j=0;j<3;j++)
    {
        if(jeu[i][j] == 1)
        {
            compteur1++;
            compteur2 = 0;

            if(compteur1 == n)
            {
                *series_j1++;
            }
        }
        else if(jeu[i][j] == 2)
        {
            compteur2++;
            compteur1 = 0;

            if(compteur2 == n)
            {
                *series_j2++;
            }
        }
    }
}
```

```

    }
}

compteur1 = 0;
compteur2 = 0;

//Verticalement
for (j=0;j<3;j++)
{
    if(jeu[j][i] == 1)
    {
        compteur1++;
        compteur2 = 0;

        if(compteur1 == n)
        {
            *series_j1++;
        }
    }
    else if(jeu[j][i] == 2)
    {
        compteur2++;
        compteur1 = 0;

        if(compteur2 == n)
        {
            *series_j2++;
        }
    }
}
}

int eval(int** jeu)
{
    int vainqueur,nb_de_pions = 0;
    int i,j;

    //On compte le nombre de pions présents sur le plateau
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            if(jeu[i][j] != 0)
            {
                nb_de_pions++;
            }
        }
    }

    if( (vainqueur = gagnant(jeu)) != 0)
    {
        if( vainqueur == 1 )
        {
            return 1000 - nb_de_pions;
        }
        else if( vainqueur == 2 )
        {
            return -1000 + nb_de_pions;
        }
        else
        {
            return 0;
        }
    }

    //On compte le nombre de séries de 2 pions alignés de chacun
    des joueurs
    int series_j1 = 0, series_j2 = 0;

```

```

        nb_series(jeu,&series_j1,&series_j2,2);

        return series_j1 - series_j2;

    }

```

Gagnant :

Code : C

```

int gagnant(int** jeu)
{
    int i,j;
    int j1,j2;

    nb_series(jeu,&j1,&j2,3);

    if(j1)
    {
        return 1;
    }
    else if(j2)
    {
        return 2;
    }
    else
    {
        //Si le jeu n'est pas fini et que personne n'a gagné,
        on renvoie 0
        for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
            {
                if(jeu[i][j] == 0)
                {
                    return 0;
                }
            }
        }

        //Si le jeu est fini et que personne n'a gagné, on renvoie 3
        return 3;
    }
}

```



Avec ce code, on remarque que les parties jouées sont toujours les mêmes. 😞

On peut ajouter une dimension aléatoire à l'algorithme sans perdre en efficacité en effectuant une petite modification au niveau des fonctions IA_jouer, Min et Max :

Dans les fonctions IA_jouer et Max:

Code : C

```

//Ligne 16 pour IA_jouer, et 20 pour Max

if(tmp > max || ( tmp == max) && (rand()%2 == 0) ) )

```

Dans la fonction Min:

Code : C

```
//Ligne 20  
  
if(tmp < min || ( tmp == min) && (rand()%2 == 0) ) )
```

En gros, si on a deux valeurs égales, on en choisira une aléatoirement.

Ne pas oublier d'initialiser la graine dans le main à l'aide de *srand* 😊.

Voilà pour ce TP. 😊

Et voilà, on est arrivés à la fin de ce tutoriel d'introduction aux IA .

J'espère qu'il vous a été utile, et vous a légèrement éclairé sur le principe des algorithmes d'intelligence artificielle.

Si vous avez des questions, vous pouvez bien sûr m'envoyer un MP.

Merci d'avoir lu mon tuto et à bientôt 😊.

Partager

