Projet d'algorithmique et complexité : MORPION

Ferhat Yousfi, Jeremy Ballot

December 22, 2014

Sommaire

Chapter 1

Introduction

Le jeu du Morpion est un jeu de réflexion se pratiquant a deux joueurs qui a tour de role ont pour but de réaliser un alignement d'un certain nombre sur une grille de taille variable. Les règles sont relativement simples, les joueurs ont un symbole qui leur est propre (en général croix ou rond), et jouent chacun leur tour sur une grille et doivent réussir a aligner dans notre cas 5 de leurs symboles horizontalement, verticalement ou en diagonale pour gagner la partie .

Chapter 2

Construction du jeu

2.1 Généralités sur le projet

Nous avons réalisé notre application en Caml. Ce langage a la particularité de pouvoir programmer en mélangeant différents type qui sont la programmation fonctionnelle, impérative (et egalement par objet, mais nous n'avons pas utilisé ce type dans notre projet). Nous avons décidé de programmer notre application en limitant un maximum la taille de chaque fonction et ainsi divisé le programme en plus de fonction faisant chacune une tache bien particulière. Ainsi notre projet sera plus clair il sera plus facile a une personne extérieure de comprendre notre code et de le modifier s'il le veut. Nous comptons gagner en rapidité et optimiser au mieux les réponses et les résultats obtenus .

2.1.1 Vue globale du programme présentation

Le programme se nomme morpi, il est décomposé en plusieurs structures qui vont permettrent d'une part un affichage graphique du morpion et par la suite de réaliser un MiniMax qui sera une fonction capable de prédire les coups de l'adversaire et ainsi jouer de la meilleure facon possible dans le but de gagner tout en bloquant l'adversaire.

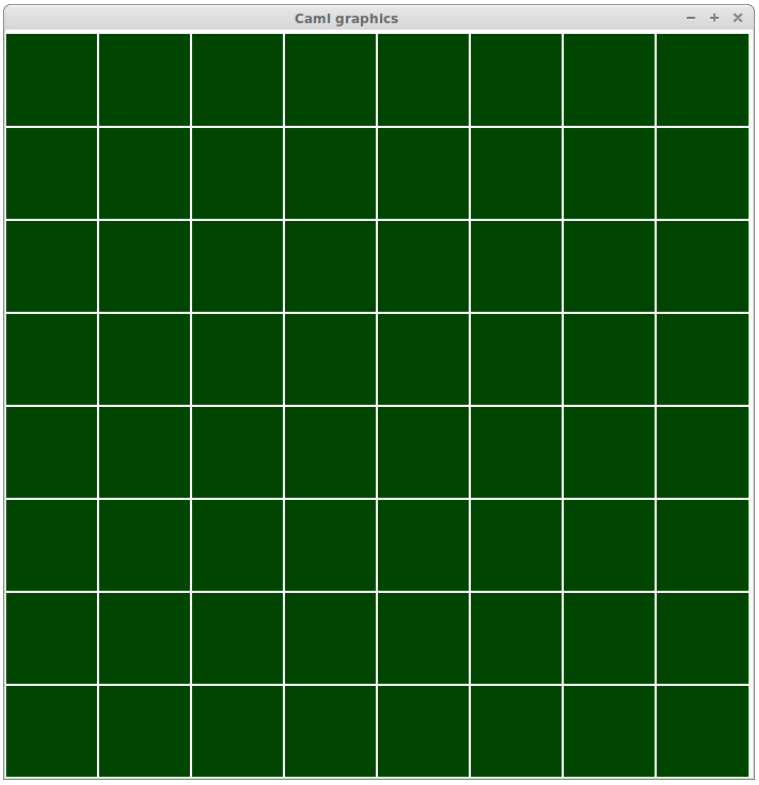
2.1.2 Visuel

Grille

Pour commencer, nous avons voulu afficher une grille pouvant contenir notre morpion pour pouvoir ensuite visualiser plus facilement les possibilités des joueurs et problèmes qui pourraient avoir lieu.

```
1 let taille_grille = ref 8
2 let taille_cell = ref (700 / !taille_grille)
3 let taille_gaine = ref 5
4
5 let dessin x grille =
6 open_graph " 700x700"; clear_graph();
7 let k=700/x in
8 for i = 0 to x-1 do
```

```
 9    for j = 0 to x-1 do
10    Graphics.set_line_width 3;
11    Graphics.set_color (Graphics.rgb 255 255 255);
12    Graphics.draw_rect(i*k)(j*k)(i+k)(j+k);
13    Graphics.set_color (Graphics.rgb 0 70 0);
14    Graphics.fill_rect(i*k+1)(j*k+1)(k-2)(k-2);
15    match grille.(i).(j) with
16      |Croix->
17        Graphics.set_color (Graphics.rgb 150 150 150);
18        Graphics.set_line_width 5;
19        des_croix i j k;
20        Graphics.set_line_width 1;
21      |Rond ->
22        Graphics.set_line_width 5;
23        Graphics.set_color (Graphics.rgb 0 150 150);
24        Graphics.draw_circle (i * k + (k/2))(j*k + (k/2))((k / 2) - 5)
25      |Vide-> Graphics.moveto 0 0;
26    done
27  done;;
```



Visuel des joueurs

Ensuite nous avons réalisé un rond et une croix que l'on pourra placer dans chaque cellule et ainsi nous pourrons différencier chaque joueur.

```
type celulle =
   Rond
  (* M\'ethode pour dessiner les Rond  *)

let dessin_cellule x =
open_graph " 512x512"; clear_graph();
let k=512/x in
for i = 0 to x do
  for j = 0 to x do
    Graphics.draw_circle (i * k + (k/2))(j * k + (k/2))((k / 2) - 5)
  done
done;;

(* M\'ethode pour dessiner les croix  *)

let des_croix i j k =
 Graphics.moveto ((i*k)+10)((j*k)+10);
 Graphics.lineto ((i*k+k)-10) ((j*k + (k))-10);
 Graphics.moveto ((i*k)+10)((j*k+k)-10);
 Graphics.lineto ((i*k+k)-10)((j*k)+10);
  ;;

  |Vide  ;;
```

2.1.3 Types

Cellule

Nous allons rentrer maintenant dans la partie qui va gérer l'affichage du morpion. Pour cela on a réalisé un type cellule, qui représente chaque case du morpion et dont son contenu peut etre soit: Vide, avec une Croix ou un Rond. La création d'une matrice a également été nécessaire qui sera de la taille de la grille affichée.

```
type celulle =
   Rond
  |Croix
  |Vide  ;;

(*Creation d'une grille vide*)
let grille = ref (Array.make_matrix !taille_grille !taille_grille Vide);;
```

2.1.4 Mise en place et fonctionnement

grille pleine

La condition d'arret evidente est quand la grille est pleine, la fonction suivante teste si toutes les cellules de la grille ont été utilisées.

```
let est_fin grille  =
let fini= ref true in
  for i=0 to (Array.length grille) -1 do
    for j=0 to (Array.length grille.(i))-1 do
      if grille.(j).(i)= Vide then fini:= false
    done;
   done;
   !fini;;
```

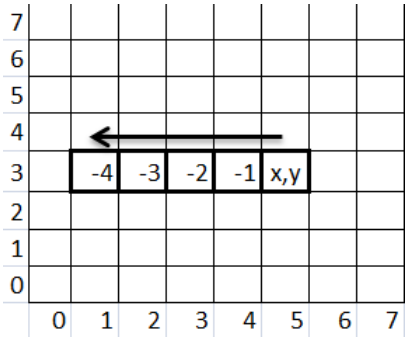
Alignement

Ce test été réalisé en différentes versions, tout d'abord nous avions enumeré toutes les possibilités, la fonction était donc longue et demandait beaucoup de tests selon la direction (horizontale, verticale...). Après de nombreux tests, nous somme arrivés a une fonction plus efficace et plus optimale qui nous donnait ce qu'on a appelé le gain.

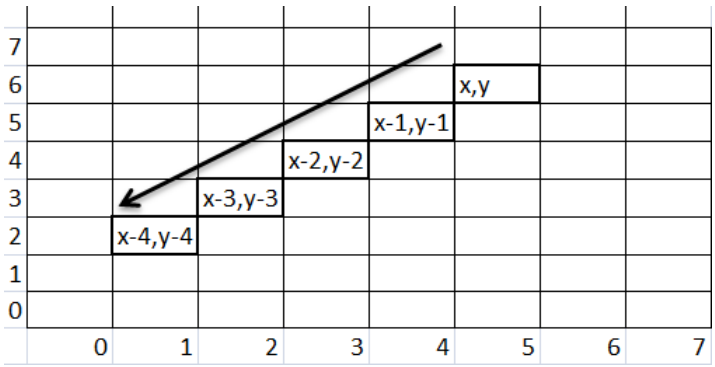
```
let verif_gain x y grille =
  let ok = ref false in
  if grille.(x).(y)=Vide then
  begin
  ok := false;
  !ok
  end
  else
  let directions = [ (*definition des direction de recherche*)
  (-1,-1, 1,1); (-1,0,1,0); (-1,1,1,-1);
  (0 ,-1,0,1) ] in
(List.iter (fun (a,b,c,d)-> if (cherche_ligne (a,b,c,d) x y grille) && !ok
    = false then ok:= true

 ) directions );
!ok;;
```

Cette fonction va donc verifier le gain d'une cellule jouée selon l'axe x et y. Nous testons si celle-ci est Vide tout d'abord (aucun gain possible) ensuite nous initialisons des directions selon x et y et donc quatre cas possibles, nous allons décrire deux d'entre eux:

- le premier cas est horizontal : (-1,0,1,0) ici la valeur de l'abscisse va diminuer de 1 pour aller sur la gauche jusqu'a ce que le point x y en paramètre soit la dernière cellule de l'alignement de cinq, de meme a droite.



- le second cas est en diagonal : (-1,-1,1,1) ici la valeur de l'absisse et ordonnée va diminuer de 1 pour aller en direction du coin en bas a gauche jusqu'a ce que le point x y en paramètre soit la dernière cellule de l'alignement de cinq; de meme en haut droite.



Cette fonction fait appel a la fonction cherche ligne, qui va par rapport aux diférentes directions trouver les bornes de la grille. Ainsi elle va appeler la fonction count line qui va calculer le nombre de signes identiques a coté de la cellule x,y se trouvant en paramètre.

```
let cherche_ligne (a,b,c,d) x y grille =
    let signe = grille.(x).(y) in
    let m =  ( init_bornes (a,b) x y  !taille_grille) in
    let i = ref (List.hd m) in
    let j = ref (List.hd(List.tl m)) in
    let n =  ( init_bornes (c,d) !i !j  !taille_grille) in
    let k = ref (List.hd n) in
    let l= ref (List.hd(List.tl n)) in

  if ((!j >=0 ) && (!i >=0 ) && (!j < !taille_grille) && (!i< !
      taille_grille)) then
    let gain = (count_line (a,b) (x+a) (y+b) !i !j signe 0 grille) +
       (count_line (c,d) (x+c) (y+d) !k !l signe 0 grille) + 1 in

    if gain < !taille_gaine then false
    else true
  else false ;;
```

La fonction suivante que l'on va vous présenter est donc la fonction count line qui va compter le nombre de symboles identiques a la suite. Ces fonctions ont été réalisées pour avoir le moins de calculs possibles, ainsi a chaque decalage dans une directon on teste les 5 cellules qui se suivent, comprenant x y et si un autre symbole est detecté il est inutile de continuer le decalage.

```
let rec count_line (a,b) x y  x' y'  signe nbr grille=
  match (a,b) with
  |(0,_)->if  (y <> (y'+b) && (x >=0 ) && (y >=0 ) && (x < !taille_grille)
  && (y < !taille_grille)) then
            if (grille.(x).(y) = signe) then
          count_line (a,b) (x+a) (y+b)  x' y' signe (nbr+1) grille
              else nbr
      else nbr

  |(_,0)-> if((x <> (x'+a)) && (x >=0 ) && (y >=0 ) && (x < !taille_grille)
  && (y < !taille_grille)) then
              if (grille.(x).(y) = signe) then
          count_line (a,b) (x+a) (y+b)  x' y' signe (nbr+1) grille
              else nbr
      else nbr

  |(_,_)->if(y<>(y'+b) && (x<> (x'+a)) && (x >=0 ) && (y >=0 )
  && (x < !taille_grille) && (y < !taille_grille)) then
              if (grille.(x).(y) = signe) then
          count_line (a,b) (x+a) (y+b)  x' y' signe (nbr+1) grille
              else nbr
    else nbr
  ;;
```

Comme vous avez pu le voir dans ces différentes fonctions nous avons utilisé un maximum de notions vues en cours pour en arriver a ce resultat.

Coups possibles

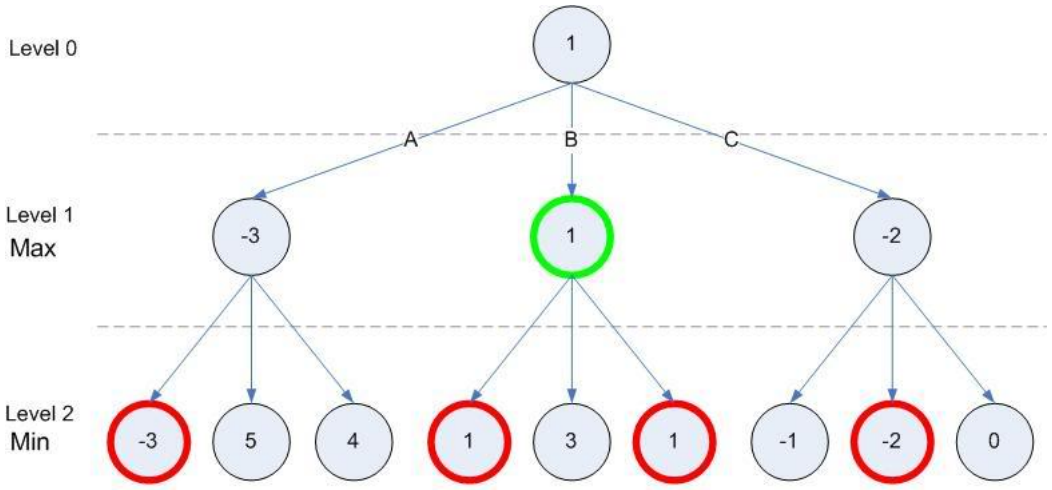
Lors d'une partie de Morpion le nombre de cases disponibles pour jouer diminue a chaque tour, il était important de savoir quel coup était disponible pour pouvoir appliquer nos recherches du meilleur coup de l'ordinateur par la suite.

```
1    let coups_jouable grille =
2   let liste_cell = ref [] in
3   for i=0 to (A.length grille)-1 do
4   for j=0 to (A.length grille.(0))-1 do
5     if (coup_legal i j grille) then
6       liste_cell := (i,j)::!liste_cell
7    done;
8    done;
9    !liste_cell;;
```

2.1.5 Le MiniMax

Gestion de la matrice

Pour pouvoir anticiper les différents coups joué par l'adversaire il est important de pouvoir jouer un coup sur la grille (matrice) et de voir ensuite les gains possibles de ce coup. La fonction MiniMax fait cela, elle peut simuler un certain nombre de coups d'avance selon la profondeur de recherche; et ainsi pouvoir anticiper les coups qui peuvent etre gagnant. Cette fonction est donc récursive et s'appelle autant de fois que la profondeur choisie.
Les conditions d'arrÃªt de cette fonction sont : soit la profondeur est atteinte soit un coup gagnant a été trouvé . Nous allons résumer ce que fait le minimax avec un schema:



Pour pouvoir faire cela il nous a semblé logique de réaliser deux fonctions qui permettent de copier une grille et simuler un coup sur cette copie de la grille.

```
let copie_grille grille =
ref (Array.init !taille_grille (fun y ->
Array.init !taille_grille (fun x -> grille.(y).(x)) ));;


let simule_coup x y signe grille=
  let grille_simule = (copie_grille grille) in
    if (coup_legal x y !grille_simule) then
      begin
      !grille_simule.(x).(y)<-signe;
      !grille_simule
      end
    else !grille_simule ;;
```

. Pour que le coup choisi soit le plus efficace il est nécessaire de donner une valeur au coup selon quel symbole gagne. La fonction qui fait cela et evalue est la suivante:

```
let evalu x y signe grille=
  let eval = ref 0 in
    if(verif_gain x y grille) then
     begin
      if ( signe ==Croix) then
      eval := ! eval-10
      else if (signe == Rond) then
      eval := ! eval+10
    end;


  !eval;;
```

Nous allons presenter la fonction minMax qui, elle, va tester tous les coups possibles. Si aucun n'est gagnant le test rappelle cette meme fonction apres avoir simul'e que l'adversaire joue un coup parmis les cases vides.

```
let rec minMax prof x y signe gril =
  let maxi = ref (-500) in
  let mini = ref (500) in
  let variable_temp = ref 0 in


  if ( prof == 0 || verif_gain x y gril ) then
    begin
    if ( verif_gain x y gril) then
      begin
      meilleurX :=x;
```

```
12        meilleurY :=y;
13        evalu x y signe gril;
14        end
15      else
16      evalu x y signe gril;
17      end
18
19    else if(prof==1 || prof== 3 ||prof == 5) then
20    begin
21      let liste_possible = coups_jouable gril in
22
23      try
24
25        (List.iter ( fun (a,b)->
26
27           let grille_test = ref (simule_coup a b Rond gril) in
28
29          variable_temp:= minMax (prof-1) a b Croix !grille_test;
30
31
32          if !mini > !variable_temp then
33
34          mini :=!variable_temp;
35
36        ))liste_possible;
37        !mini;
38        with | _ -> !mini;
39    end
40    else begin
41      let liste_possible = coups_jouable gril in
42      try
43
44        (List.iter ( fun (a,b)->
45           let grille_t = ref (simule_coup a b Croix gril) in
46          variable_temp:= minMax (prof-1) a b Rond !grille_t;
47
48          if !maxi < !variable_temp then maxi :=!variable_temp;
49        )
50      )liste_possible;
51      !maxi;
52      with | _ -> !maxi;
53    end;;
```

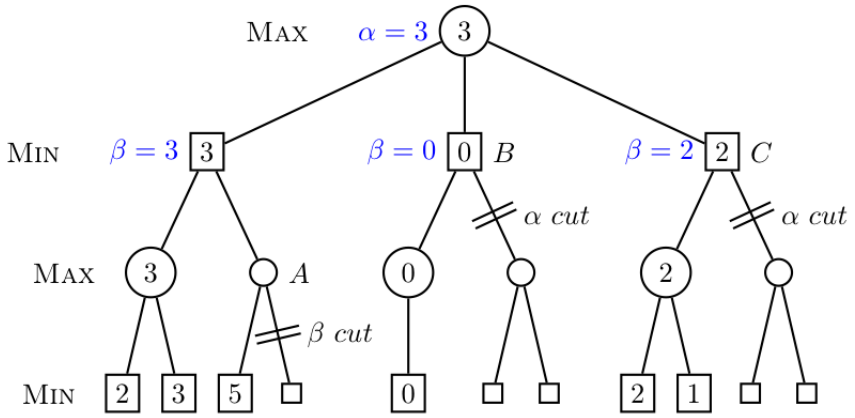
Comme on peut le voir cette fonction renvoie l'evaluation la plus grande (mini ou maxi); qui est la plus grande possibilité pour gagner de l'ordinateur ou pour l'adversaire. Il est donc necessaire de récupérer la position du point a jouer apres ces tests. La fonction réalisant cela est ordIA qui va mettre a jour les valeurs du meilleurX et meilleurY qui sera le meilleur coup a jouer selon X et Y.

```
let ordIA grille  =

    let max = ref (-1000)
    and temp = ref (0) in
    let liste_possible = coups_jouable grille in

      (List.iter ( fun (a,b)->
      let grille_t = ref (simule_coup a b Croix grille) in
      temp :=minMax 3 a b Rond !grille_t;
      if !max < !temp then
      begin
       max  :=!temp;
      meilleurX :=a;
      meilleurY :=b;

      (*Printf.printf" a,b, meilleurX, meilleurY --> (%d,%d,%d,%d) \n"a b !
          meilleurX !meilleurY*)
      end

    )
    )liste_possible;
maj_cell !meilleurX !meilleurY Croix grille;;
```

Comme on peut le voir cette fonction met a jour directement la grille en appelant la fonction une fois la boucle sur les coups possibles terminée.

2.2 MiniMax alpha-beta

Cet algo ressemble beaucoup au Minimax de base sauf que celui-ci a la particularité de couper certaine branche de l'arbre lors de la recherche. Une comparaison est faite selon la valeur des feuilles, ainsi les valeurs les plus grandes ou plus petites sont gardées selon si on calcule le min ou le max. et la branche qui ne satisfait pas cette condition est suprimée. Cela permet de gagner du temps lors de l'algorithme car il teste moins de valeurs. Voici un schema qui devrait expliquer cela:



```
let mini x y prof (alpha:int ref) (beta:int ref) grille =
  let temp = ref 0 in
  if ( prof == 0 || verif_gain x y grille) then
    evalu2 x y grille
  else
  begin
    let liste_possible = coups_jouable grille in
    try
      (List.iter  (fun (a,b)->
        grille.(a).(b)<- Croix;
        temp := maxi a b (prof-1) alpha beta grille;
        grille.(a).(b)<-Vide;
        if !beta > !temp then beta := !temp;
        if !beta <= !alpha then raise Exit;
        )
      )liste_possible;
      !beta;
      with | _ -> !beta;
  end;;

let maxi x y prof (alpha:int ref) (beta:int ref) grille =
  let temp = ref 0 in
  if ( prof == 0 || verif_gain x y grille) then
    evalu2 x y grille
  else
  begin
    let liste_possible = coups_jouable grille in

    try

      (List.iter  (fun (a,b)->

        grille.(a).(b)<- Rond;
        temp := mini a b (prof-1) alpha beta grille;
        grille.(a).(b)<-Vide;
        if !alpha > !temp then beta := !temp;
        if !alpha <= !alpha then raise Exit;
        )
      )liste_possible;
      !beta;
      with | _ -> !beta;
  end;;
```

Chapter 3

Expérimentation et test

3.1 profondeur

La profondeur est essentielle pour avoir un ordinateur plus efficace.

- En effet une profondeur 0 permet essentiellement a l'ordinateur de voir si il peut gagner ou non.
- La profondeur 1 permet de voir si l'adversaire va gagner ou pas au prochain tour et ainsi le bloquer.
- la profondeur 2 permet a l'ordinateur de predire que s'il a 3 cases alignées il peut dans 2 coups gagner la partie.
- la profondeur 3 permet de prédire que dans 3 coups le joueur adverse peut gagner et donc le bloquer.

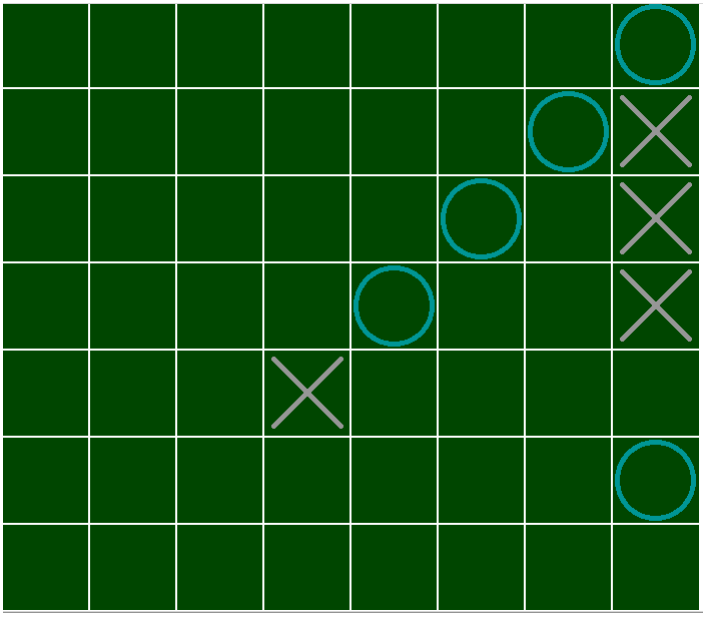
Nous avons fait de nombreux tests qui nous ont permis d'avoir des résultats que nous allons vous presenter en images:

```
# grille;;
- : celulle array array ref =
{contents =
  [|[|Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Croix; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Croix; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Croix|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Vide|]|]}
```

Ici nous avons delibérement placé 3 croix alignées et appliqué le MiniMax avec une profondeur de 2. L'ordinateur joue donc sur l'une des 2 cases entourant l'alignement de 3.

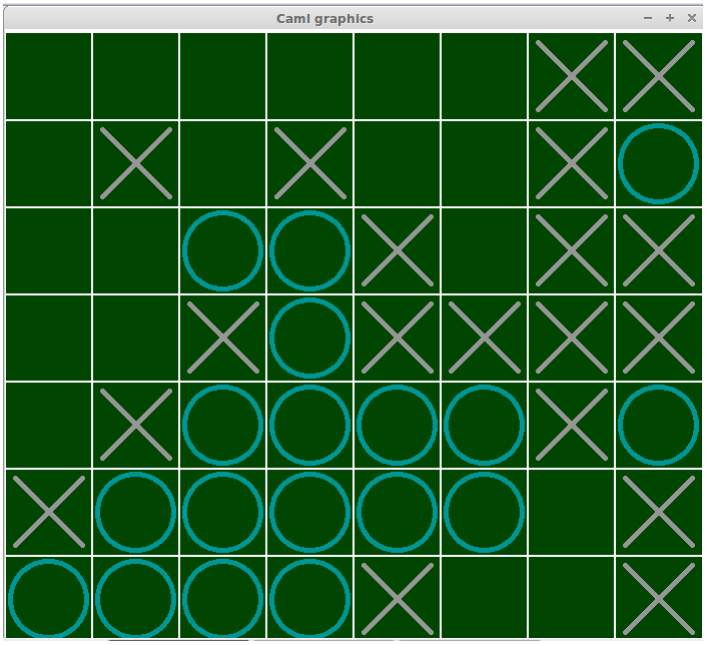
```
# grille;;
- : celulle array array ref =
{contents =
  [|[|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Croix; Vide; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Croix; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Croix; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Croix; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
    [|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|]|]}
```

Comme nous pouvons le voir l'ordinateur a bien joués pour forcer l'adversaire a le bloquer au prochain tour.
Nous allons maintenent voir graphiquement les coups joué par l'adversaire.



Comme on peut le voir l'ordinateur bloque bien nos différents coups lorsque nous avons la possibilité de gagner.

Nous allons maintenant voir graphiquement les coups joués par l'adversaire. Voici un exemple de partie jouée contre l'ordinateur, ou nous pouvons voir que quand il peut aligner 5 symboles ou nous bloquer, il le fait. (l'ordinateur est representé par les Croix).



Dans cet exemple nous avons joué contre un Minimax de profondeur 2.

Chapter 4

Conclusion

Ce projet nous a permis de mieux comprendre comment coder en caml et les interactions possibles avec une fenetre graphique. Le manque parfois d'information lors d'erreur a la compilation a un peu compliqué les choses. Ce projet nous a permis de revoir tout ce que l'on a vu durant l'année et plus encore.

Chapter 5

Source

- http://blog.xebia.fr/2011/09/28/lalgorithme-minimax
- http://openclassrooms.com/courses/l-algorithme-min-max
- http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML/