

1.) Les arbres binaires de recherche.	2
2.) Le tri maximier.	3
2.1.) Principe.	3
2.2.) Implémentation.	5
3.) Algorithme du MinMax.	6
3.1.) Cadre.	6
3.2.) Mini Max.	6
3.3.) Elagage a-b.	7
3.4.) Jeu « des allumettes ».	8
3.5.) Jeu de « Kono ».	10
3.6.) Jeu de « tic-tac-toe ».	17
4.) Programmation des jeux d'échecs.	24
4.1.) Introduction.	24
4.2.) Représentation des états possibles du jeu.	24
4.2.1.) Codage en 2 dimensions :	25
4.2.2.) Codage sur un tableau à 1 dimension :	27
4.3.) Liste des positions accessibles.	27
4.4.) Les algorithmes d'évaluation statique de la position.	28
4.4.1.) Dénombrement.	28
4.4.2.) Analyse de la configuration.	29
4.5.) Utilisation du MiniMax.	29
4.5.1.) Profondeur 1.	29
4.5.2.) Profondeur 2.	29
4.5.3.) Limites du MiniMax.	30
4.5.4.) Autres algorithmes.	31
5.) Exercices.	33
5.1.) Exercices.	33
Exercice 9.1 :	33
Exercice 9.2.	33
Exercice 9.3.	33
Exercice 9.4.	33

*«Les hommes n'ont jamais montré tant de sagacité que dans l'invention des jeux.»
Leibniz.*

1.) Les arbres binaires de recherche.

Un arbre binaire de recherche est un arbre binaire parfait qui possède la propriété suivante :

L'étiquette de chaque nœud est supérieure à celle de tout nœud de son sous-arbre gauche et inférieure à celle de chaque nœud de son sous-arbre droit.

2.) Le tri maximier.

2.1.) Principe.

Cette méthode de tri (aussi appelée *tri par tas* ou *heapsort*) est basée sur la notion de *tas*. Un *tas* est un arbre binaire parfait tel que l'information contenue dans tout nœud est supérieure à celle contenue dans ses fils.


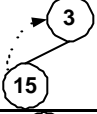
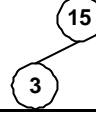
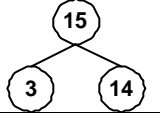
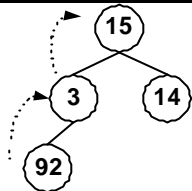
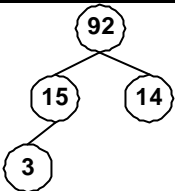
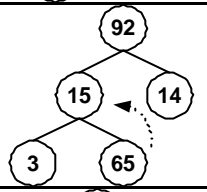
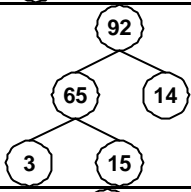
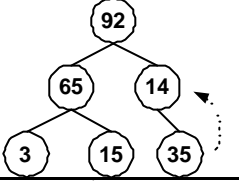
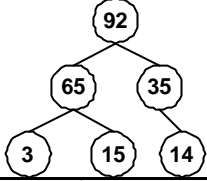
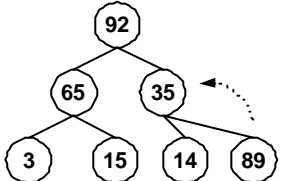
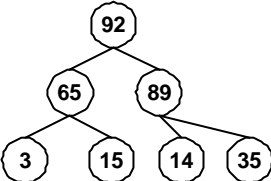
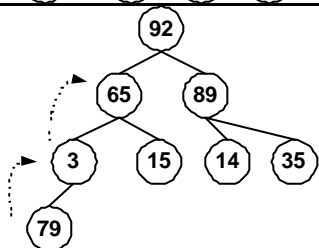
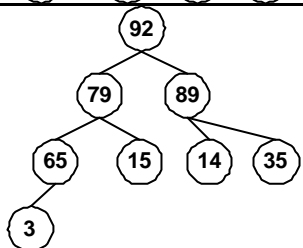
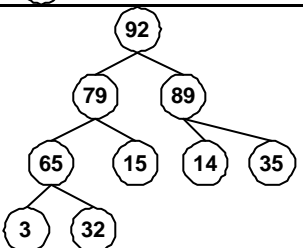
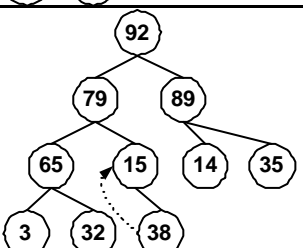
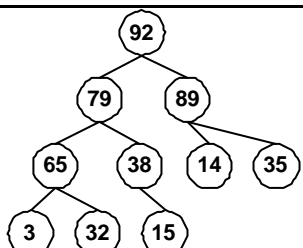
La méthode du tri par tas comporte deux étapes :

1. Construction par insertions successives d'un tas à partir du vecteur à trier.
2. On remplace la racine, qui est nécessairement le plus grand élément du tableau par son fils le plus grand. La racine est transférée à la place de la dernière feuille de l'arbre et celle-ci est repositionnée. On réitère avec la nouvelle racine autant de fois qu'il y a de nœuds dans l'arbre.

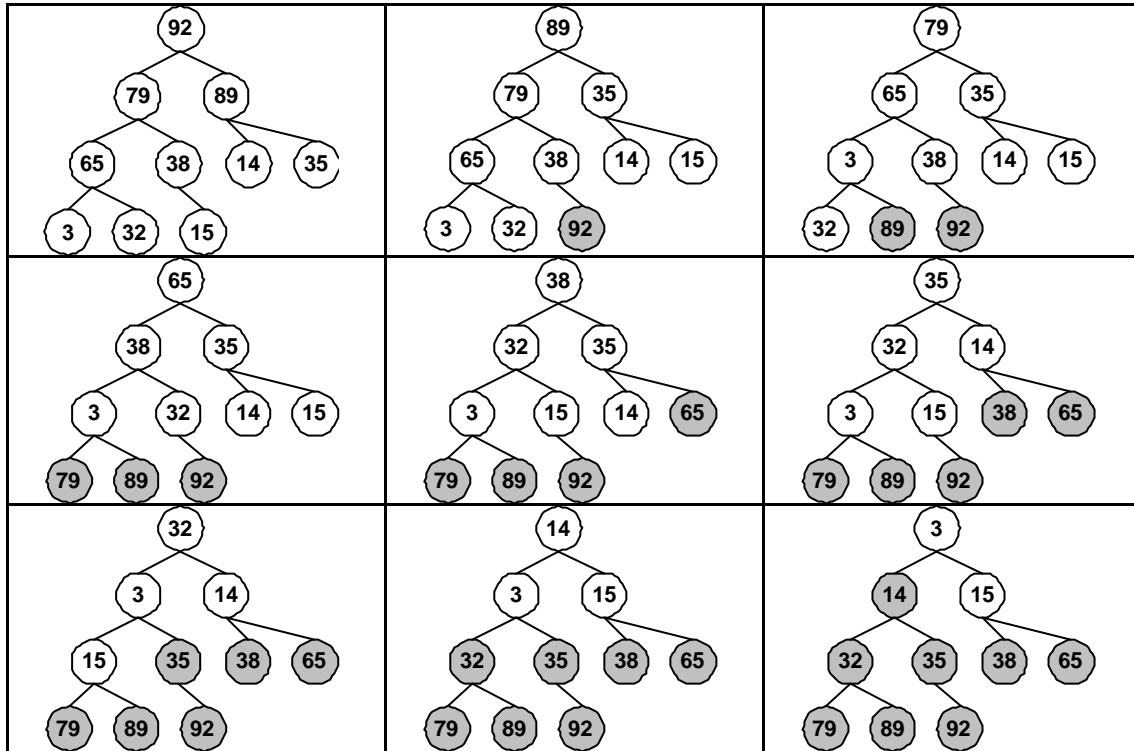
Exemple du tri par tas avec 3 ; 14 ; 15 ; 92 ; 65 ; 35 ; 89 ; 79 ; 32 ; 38 :

On commence par élaborer un tas sachant que l'arbre construit doit être parfait. Chaque élément est donc inséré en dernière position le plus à gauche possible du dernier niveau et, s'il est plus grand que son père, ils sont inversés ; on répète cette inversion autant de fois que nécessaire.

Le tableau suivant détaille chaque étape de cette première phase.

Phase	Insertion d'un élément	Tas réorganisé
On insère l'élément 3.		
On insère 15 le plus à gauche possible puis on réorganise le tas.		
On insère 14.		
On insère 92 et on réorganise le tas.		
On insère 65.		
On insère 35.		
On insère 89.		
On insère 79.		
On insère 32. Aucune réorganisation est nécessaire.		
On insère 38 et on réorganise. La première phase est terminée.		

Les figures suivantes montrent les différentes étapes de la seconde phase : On insère la racine (92) à la place du dernier nœud de l'arbre (15). La racine est ensuite remplacée par son fils le plus grand (89) lequel est remplacé par son fils le plus grand (35) etc... Le nœud qui a été écrasé par la racine (15) est inséré à la place de l'ancienne position du dernier nœud déplacé (35). On réitère avec la nouvelle racine (89) puis la suivante (79) jusqu'à obtenir le dernier arbre.



On remarque que le parcours en largeur de ce dernier arbre produit la suite triée :

3 ; 14 ; 15 ; 32 ; 35 ; 38 ; 65 ; 79 ; 89 ; 92

2.2.) Implémentation.

Si un arbre binaire est représenté par une suite x_n , alors un élément x_i donné a pour fils les éléments x_j et x_{j+1} avec $j = 2i$ et tout élément x_n (avec $n > 2$) a pour père x_k tel que $k = n \text{ div } 2$.

On voit donc qu'il est facile, disposant d'un vecteur que l'on assimile à un arbre binaire d'accéder au fils et au père de chaque élément et d'implémenter la méthode du tri par tas avec une telle structure.

3.) Algorithme du MinMax.

Le principe de base qui régit la plupart des algorithmes de jeu d'échecs et autres jeux de réflexion est la théorie du « minimax », établie en 1928 par John VON NEUMANN. Elle amène l'ordinateur à passer en revue toutes les possibilités pour un nombre limité de coups et à leur assigner une valeur qui prenne en compte les bénéfices pour le joueur et pour son adversaire. Le meilleur choix était alors celui qui maximise ses bénéfices tout en minimisant ceux de son adversaire.

3.1.) Cadre.

On considère un jeu se jouant à deux personnes, chaque joueur effectuant tour à tour un unique coup. On supposera qu'aucune partie ne comporte un nombre infini de coups, et qu'à chaque position, il existe un nombre fini de coups légaux. Alors, d'après le *lemme de l'infini*, à chaque position p , il existe un nombre fini $N(p)$ tel que toute partie commençant à partir de p ne comporte pas plus de $N(p)$ coups.

Des exemples typiques de jeux entrant dans le cadre précédent sont les jeux des Echecs, d'Othello, etc..... Le but est de construire et d'étudier des algorithmes qui, étant donnée une position p , et un joueur, choisissent un coup "valable" pour ce joueur.

Définition 1.1 : On appelle Arbre de Jeu ou Arbre de décision construit à partir de la position p l'arbre vérifiant les propriétés suivantes :

- la racine de l'arbre est la position p .
- Soit n un nœud de l'arbre; si n est terminal alors c'est une feuille. Sinon si c_1, \dots, c_d sont les coups possibles légaux à partir de la position n , donnant les positions n_1, \dots, n_d , alors n a pour fils n_1, \dots, n_d .

Ceci définit clairement par récurrence un arbre, qui est fini d'après le *Lemme de l'infini*. C'est à partir de cet arbre que l'on construira et que l'on étudiera les algorithmes choisissant un coup à partir de la position p .

Soit p une position terminale. A l'aide d'une *fonction d'évaluation*, on associe une *valeur* à cette position, valeur que l'on notera par la suite $f(p)$. Par exemple, cela peut être +1 si la partie se termine par une victoire pour le joueur que le programme représente, -1 pour une défaite, et 0 pour un nul. C'est à partir de ces valeurs assignées aux nœuds terminaux que l'algorithme déterminera le coup choisi.

Dans la pratique, pour des contraintes de temps et d'espace, on ne peut explorer un arbre de jeu dans sa totalité. On décide alors que les nœuds à une profondeur d de l'arbre sont terminaux. A tous ces nœuds, la *fonction d'évaluation* associe une estimation de la valeur de la position à l'aide de critères plus ou moins subjectifs. Par exemple, pour les échecs, l'estimation peut se baser sur le nombre de pièces de part et d'autre, sur la mobilité, l'avance de développement, etc.

Les problèmes à résoudre sont les mêmes que dans le cas théorique idéal. Les contraintes de temps prennent une importance capitale ; améliorer le temps de parcours de l'arbre signifie en effet pouvoir explorer une profondeur plus élevée, et ainsi jouer mieux.

3.2.) Mini Max.

L'algorithme *Mini Max*, 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 :

- $MiniMax(p) = f(p)$ si p est une position terminale
- $MiniMax(p) = \max(MiniMax(o_1), \dots, MiniMax(o_n))$ si p est une position joueur avec fils o_1, \dots, o_n
- $MiniMax(p) = \min(MiniMax(j_1), \dots, MiniMax(j_m))$ si p est une position opposant avec fils j_1, \dots, 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 $h : position \rightarrow \mathbf{R} \cup \{\pm\infty\}$ 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 prend une position p et retourne la liste des positions accessibles par les coups disponibles à partir de p .

3.3.) Elagage α - β

L'algorithme α - β est une optimisation du Mini Max, qui «coupe» des sous arbres dès que leur valeur devient inintéressante aux fins du calcul de la valeur Mini Max 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 :

```

FONCTION ALPHA-BETA(P, A, B) (ici A est toujours inférieur à B)
DEBUT
  SI P est une feuille ALORS RETOURNER la valeur de P
  SINON
    INITIALISER Alpha de P à  $-\infty$  et Beta de P à  $+\infty$ ;
    SI P est un noeud 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 RETOURNER de P
      FIN POUR TOUT FAIRE
      RETOURNER 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 RETOURNER Alpha de P
      FIN POUR TOUR FAIRE
      RETOURNER Alpha de P;
  FIN ALPHA-BETA;

```

On sait que la véritable valeur Mini Max 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 Mini Max(P).

ALPHA-BETA permet assez souvent de doubler la profondeur d'exploration d'un arbre à parité de ressources, par rapport à Mini Max.

3.4.) Jeu « des allumettes ».

On considère le jeu suivant : Au départ on dispose quelques allumettes sur une table. Chaque joueur peut choisir de prendre 1,2 ou 3 allumettes en même temps. Le perdant est celui qui retire la dernière allumette.

4																							
3						2						1+											
2	1-		0			1-	0		0			0	0		0								
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
+																							

```

#let rec evalue = fun
| n x when n>3 -> evalue (n-1) (not x) + evalue (n-2) (not x) +
evalue(n-3) (not x)
| 3 x -> evalue (2) (not x) + evalue (1) (not x)
| 2 x -> evalue (1) (not x)
| 1 true -> -1
| 1 false -> 1
| _ _ -> failwith "erreur";;
evalue : int -> bool -> int = <fun>

```

Voilà le jeu complet :

```
let meilleur = fun
```



```

| a b c when (a>=b) && (a>=c)-> 1
| a b c when (b>=a) && (b>=c)->2
| _ _ c -> 3;;

let jeu n =
let j = ref n and i = ref 0 and s = ref " " in
while !j > 0 do

  if !j > 3 then i := meilleur (evaluate (!j - 1) false) (evaluate
  (!j - 2) false) (evaluate (!j - 3) false)
  else if !j=3 then i:=2 else i:=1;

  print_string "Je prend ";
  print_int !i;
  print_string " allumette(s). Il en reste ";
  j := !j - !i;
  print_int !j;
  print_newline();
  if !j=0 then print_string "Bravo vous avez gagné ! "
  else
  begin
    print_string "Donnez votre jeu ";
    s:=read_line();
    i:=int_of_string !s;
    print_newline();
    j := !j - !i;
    if !j = 0 then print_string "Vous avez perdu ! " else ();
  end
end;
done;;

```

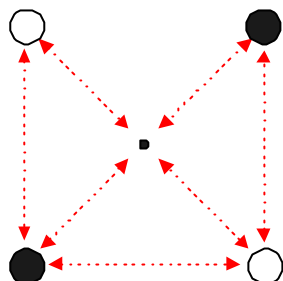
Voici le déroulement du jeu en partant avec 20 allumettes :

Commentaires	Déroulement du jeu
Lancement du jeu	#jeu 20;;
Coup de l'ordinateur.	Je prend 3 allumette(s). Il en reste 17
Coup du joueur	Donnez votre jeu : 2
Coup de l'ordinateur	Je prend 2 allumette(s). Il en reste 13
Coup du joueur	Donnez votre jeu : 3
Coup de l'ordinateur	Je prend 1 allumette(s). Il en reste 9
Coup du joueur	Donnez votre jeu : 1
Coup de l'ordinateur	Je prend 3 allumette(s). Il en reste 5
Coup du joueur	Donnez votre jeu : 2
Coup de l'ordinateur	Je prend 2 allumette(s). Il en reste 1
Coup du joueur	Donnez votre jeu : 1
Résultat du jeu.	Vous avez perdu ! - : unit = ()

Bien entendu, plus le nombre d'allumettes est grand, plus longue est l'exploration de l'arbre. Avec 20 allumettes, ce temps devient déjà nettement perceptible. Essayez avec 30 allumettes !

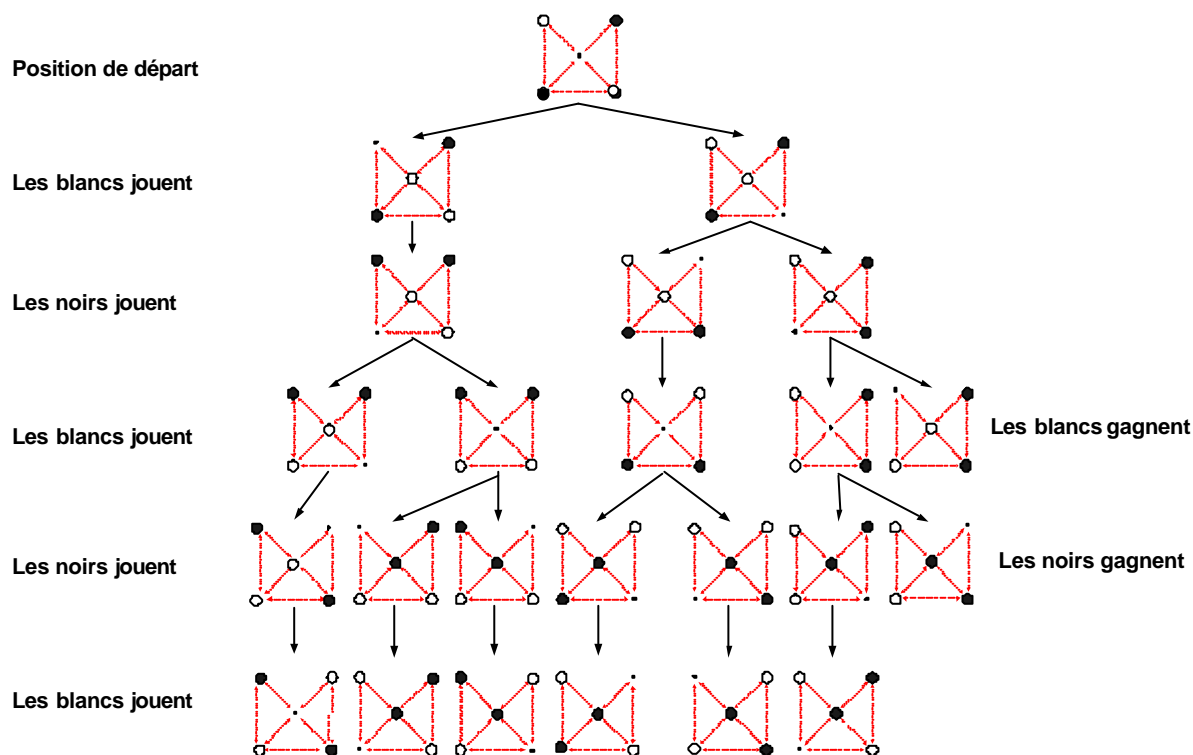
3.5.) Jeu de « Kono ».

Le jeu de « Kono » est un jeu joué dans tout l'extrême orient, avec quelques variations. C'est un jeu assez trivial surtout pratiqué par des enfants. Il se joue sur un plateau ne contenant que 5 points reliés de la manière suivante :



Chaque joueur a deux pièces qui restent en jeu durant toute la partie. Les positions de départ sont illustrées ci-dessus. En bougeant alternativement une pièce sur un point adjacent, le joueur tente de piéger son adversaire pour lui empêcher tout mouvement.

L'arbre correspondant aux mouvements possibles est illustré ci-dessous :



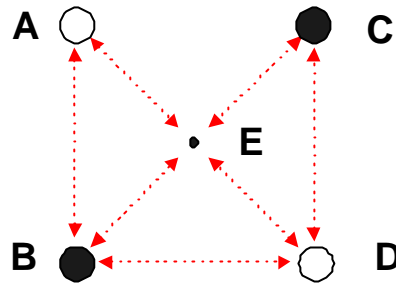
Techniquement, le jeu est nul, car en jouant correctement, il est toujours possible d'éviter la défaite et l'arbre est donc infini.

On pourrait implémenter ce jeu en Caml de la manière suivante :

Le type somme « Situation » permet de modéliser les états possibles des cases du plateau :

```
#type Situation = Noir | Blanc | Vide;;
Le type Situation est défini.
```

On modélise ensuite le plateau par le n-uplet (a,b,c,d,e) dont chaque composante représente une case du plateau :



(a,b,c,d,e) est donc du type « Situation * Situation * Situation * Situation * Situation ».

La fonction « transitions_possibles » permet, à partir d'une position donnée de lister toutes les positions possibles :

```
#let transitions_possibles = fun
| (Vide,Vide,Vide,Vide,Vide) _ -> failwith "Erreur"
| (a,b,c,d,e) couleur -> let l = ref [] in
begin
  if a=couleur then
  begin
    if b=Vide then l := ((Vide,a,c,d,e) :: !l);
    if e=Vide then l := ((Vide,b,c,d,a) :: !l);
  end;
  if b=couleur then
  begin
    if a=Vide then l := ((b,Vide,c,d,e) :: !l);
    if d=Vide then l := ((a,Vide,c,b,e) :: !l);
    if e=Vide then l := ((a,Vide,c,d,b) :: !l);
  end;
  if c=couleur then
  begin
    if d=Vide then l := ((a,b,Vide,c,e) :: !l);
    if e=Vide then l := ((a,b,Vide,d,c) :: !l);
  end;
  if d=couleur then
  begin
    if b=Vide then l := ((a,d,c,Vide,e) :: !l);
    if c=Vide then l := ((a,b,d,Vide,e) :: !l);
    if e=Vide then l := ((a,b,c,Vide,d) :: !l);
  end;
  if e=couleur then
  begin
    if a=Vide then l := ((e,d,c,d,Vide) :: !l);
    if b=Vide then l := ((a,e,c,d,Vide) :: !l);
    if c=Vide then l := ((a,b,e,d,Vide) :: !l);
    if d=Vide then l := ((a,b,c,e,Vide) :: !l);
  end;
end;
!l;;
transitions_possibles :
```

```

Situation * Situation * Situation * Situation * Situation ->
Situation ->
(Situation * Situation * Situation * Situation * Situation) list =
<fun>

```

La fonction « transitions_possibles » reçoit en paramètre une position initiale et une couleur et renvoie une liste de toutes les positions possibles à partir de la position initiale. Voici, par exemple, les transitions possibles à partir de la position initiale si on joue avec les blancs :

```

#transitions_possibles (Noir,Blanc,Blanc,Noir,Vide) Blanc;;
- : (Situation * Situation * Situation * Situation * Situation) list =
[Noir, Blanc, Vide, Noir, Blanc; Noir, Vide, Blanc, Noir, Blanc]

```

et si on joue avec les noirs :

```

#transitions_possibles (Noir,Blanc,Blanc,Noir,Vide) Noir;;
- : (Situation * Situation * Situation * Situation * Situation) list =
[Noir, Blanc, Blanc, Vide, Noir; Vide, Blanc, Blanc, Noir, Noir]

```

Il nous faut encore une fonction permettant d'afficher les positions .:

```

#let affiche_position (a,b,c,d,e) =
begin
  print_newline();
  imprime a;print_string "      ";imprime c;print_newline();
  print_string "      ";imprime e;print_newline();
  imprime b;print_string "      ";imprime d;print_newline();
end
where imprime = fonction
| Noir -> print_string "N"
| Blanc -> print_string "B"
| _ -> print_string "." ;;
affiche_position :
Situation * Situation * Situation * Situation * Situation -> unit =
<fun>

```

Ce qui nous donne, par exemple :

```

#affiche_position (Blanc,Noir,Noir,Blanc,Vide);;

B      N
      .
N      B
- : unit = ()

```

Une fonction permettant au joueur « humain » de rentrer son jeu. Il rentrera son jeu en indiquant la nouvelle position dans l'ordre des cases ACEBD et en notant N pour un pion noir, B pour un blanc et V pour une case vide. Par exemple, la situation initiale se notera « BNVNB » :

```

#let donne_position () =
  print_newline();
  print_string "Donnez votre jeux (N=noir,B=Blanc,V=Vide) :";
  let donne_valeur = fonction

```

```

| `N` -> Noir
| `B` -> Blanc
| `V` -> Vide
| _ -> failwith "Erreur"
and s = read_line() in
let a = donne_valeur s.[0]
and c = donne_valeur s.[1]
and e = donne_valeur s.[2]
and b = donne_valeur s.[3]
and d = donne_valeur s.[4]
in (a,b,c,d,e)
;;
donne_position :
unit -> Situation * Situation * Situation * Situation * Situation =
<fun>

```

Une fonction permettant d'évaluer la valeur d'un coup. On notera +10 un jeu gagnant, -1 un jeu perdant et 0 un jeu nul. On limitera la profondeur de la recherche à 6 niveaux. Cette fonction reçoit en paramètre une liste de positions, la profondeur de recherche, le joueur dont c'est le tour (Noir ou Blanc) et le joueur pour qui on réalise l'évaluation (Noir ou Blanc) :

```

#let rec evalue = fun
| [] _ tour couleur when tour=couleur ->(-5)
| [] _ tour couleur when tour<>couleur -> 10
| _ 0 _ _-> 0
| [a] n Blanc couleur -> let l = transitions_possibles a Noir in
  evalue l (n-1) Noir couleur
| [a] n Noir couleur -> let l = transitions_possibles a Blanc in
  evalue l (n-1) Blanc couleur
| (t::q) n tour couleur -> (evalue [t] n tour couleur) +
  (evalue q n tour couleur)
| _ _ _ _ -> failwith "Erreur";;
evalue :
(Situation * Situation * Situation * Situation * Situation) list ->
int -> Situation -> Situation -> int = <fun>

```

Une fonction qui, à partir d'une position donnée, me renvoie le meilleur coup possible :

```

#let meilleur_coup (a,b,c,d,e) couleur =
  let rec meilleur_score = fun
    | [] _ _ -> failwith "J'ai perdu la partie !!!"
    | [a] s o -> let n = (evalue [a] 6 couleur couleur) in if n>=s
then a else o
    | (t::q) s o -> let n = (evalue [t] 6 couleur couleur) in
      if n>=s then meilleur_score q n t else meilleur_score q s o
  in let l = transitions_possibles (a,b,c,d,e) couleur in
  let s = meilleur_score l (-20) (a,b,c,d,e) in s ;;
meilleur_coup :
Situation * Situation * Situation * Situation * Situation ->
Situation -> Situation * Situation * Situation * Situation *
Situation =
<fun>

```

Et enfin la fonction principale du jeu :

```

#let jeu couleur =

```

```

let p = ref (Blanc,Noir,Noir,Blanc,Vide) and l = ref [] and
adversaire = if couleur=Blanc then Noir else Blanc
in
begin
  if couleur = Noir then print_string "Vous avez les Blancs "
  else print_string "Vous avez les noirs ";
  print_newline();
  affiche_position !p;
  while true do
    p := donne_position();
    affiche_position !p;
    p := meilleur_coup !p couleur;
    print_string "Je joue :";
    print_newline();
    affiche_position !p;
    l:=transitions_possibles !p adversaire;
    if (list_length !l)=0 then failwith "Vous avez perdu !";
  done;
end;;
jeu : Situation -> unit = <fun>

```

On lance le jeu en appelant la fonction jeu et lui passant en paramètre la couleur avec laquelle l'ordinateur joue. Par exemple :

```
#jeu Blanc;;
```

CamL répond en donnant la couleur de son adversaire et la situation initiale :

```

Vous avez les noirs

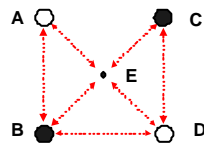
  B      N
    .
  N      B

```

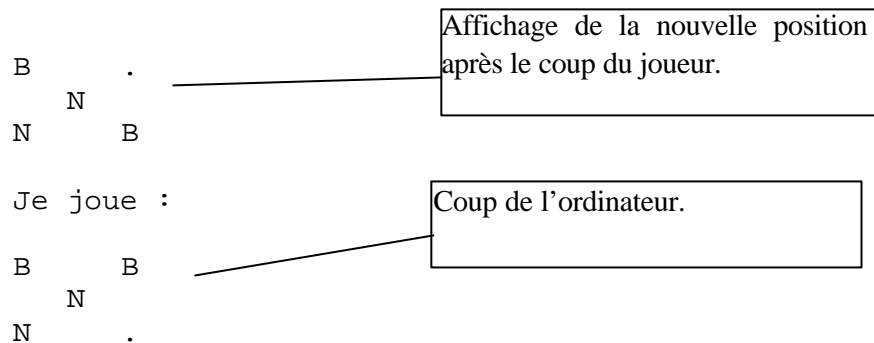
Puis, la fonction entre dans une boucle dans laquelle CamL demande le coup du joueur « humain » :

```
Donnez votre jeux (N=noir,B=Blanc,V=Vide) :
```

On entre son jeu sous la forme d'une chaîne de 5 lettres en notant N pour un pion noir, B pour un pion blanc et V pour une case vide dans l'ordre des sommets ACEBD :



On jouera par exemple BVNNB. Après chaque coup, CamL affiche la nouvelle position puis donne le coup de l'ordinateur :



Le point « . » indique une case vide, « B » un pion blanc et « N » un pion noir.

Ensuite, soit la partie continue et Caml demande le coup du joueur « humain » soit la partie s'interrompt sur une victoire de l'ordinateur ou (plus rarement) du joueur.

```

CAML for Windows - [Terminal]
File Edit Caml Display Help
#jeu Blanc;;
Vous avez les noirs
B   N
  .
N   B
Donnez votre jeux (N=noir,B=Blanc,V=Vide) :BNVVB
B   N
  N
.   B
Je joue :
B   N
  N
B   .
Donnez votre jeux (N=noir,B=Blanc,V=Vide) :
BNVBB
For Help, press F1

```

Figure 1 Jeu de "Kono".

Pour résumer, le programme complet :

```

type Situation = Noir | Blanc | Vide;;

let transitions_possibles = fun
(* Transitions possibles à partir d'une position *)
| (Vide,Vide,Vide,Vide,Vide) _ -> failwith "Erreur"
| (a,b,c,d,e) couleur -> let l = ref [] in
begin
  if a=couleur then
  begin
    if b=Vide then l := ((Vide,a,c,d,e) :: !l);
    if e=Vide then l := ((Vide,b,c,d,a) :: !l);
  end;
  if b=couleur then
  begin
    if a=Vide then l := ((b,Vide,c,d,e) :: !l);
    if d=Vide then l := ((a,Vide,c,b,e) :: !l);
    if e=Vide then l := ((a,Vide,c,d,b) :: !l);
  end;
  if c=couleur then
  begin
    if d=Vide then l := ((a,b,Vide,c,e) :: !l);
    if e=Vide then l := ((a,b,Vide,d,c) :: !l);
  end;
  if d=couleur then
  begin
    if b=Vide then l := ((a,d,c,Vide,e) :: !l);
    if c=Vide then l := ((a,b,d,Vide,e) :: !l);
    if e=Vide then l := ((a,b,c,Vide,d) :: !l);
  end;
  if e=couleur then
  begin
    if a=Vide then l := ((e,d,c,d,Vide) :: !l);
    if b=Vide then l := ((a,e,c,d,Vide) :: !l);
    if c=Vide then l := ((a,b,e,d,Vide) :: !l);
    if d=Vide then l := ((a,b,c,e,Vide) :: !l);
  end;
end;
!l;;

let affiche_position (a,b,c,d,e) =
(* Affichage de la position passée en paramètre *)
begin
  print_newline();
  imprime a;print_string " ";imprime c;print_newline();
  print_string " ";imprime e;print_newline();
  imprime b;print_string " ";imprime d;print_newline();
end
where imprime = fonction
| Noir -> print_string "N"
| Blanc -> print_string "B"
| _ -> print_string "." ;;

let donne_position () =
(* Lecture du coup du joueur humain *)
print_newline();
print_string "Donnez votre jeux (N=noir,B=Blanc,V=Vide) : ";
let donne_valeur = fonction
| `N` -> Noir
| `B` -> Blanc
| `V` -> Vide
| _ -> failwith "Erreur"
and s = read_line() in
let a = donne_valeur s.[0]
and c = donne_valeur s.[1]
and e = donne_valeur s.[2]
and b = donne_valeur s.[3]
and d = donne_valeur s.[4]
in (a,b,c,d,e)
;;

```



```

let rec evalue = fun
(* Evaluation d'un coup *)
| [] _ tour couleur when tour=couleur ->(-5)
| [] _ tour couleur when tour<>couleur -> 10
| _ 0 _-> 0
| [a] n Blanc couleur -> let l = transitions_possibles a Noir in
  evalue l (n-1) Noir couleur
| [a] n Noir couleur -> let l = transitions_possibles a Blanc in
  evalue l (n-1) Blanc couleur
| (t::q) n tour couleur -> (evalue [t] n tour couleur) +
  (evalue q n tour couleur)
| _ _ _ _-> failwith "Erreur";;

let meilleur_coup (a,b,c,d,e) couleur =
(* Recherche du meilleur coup possible à partir d'une position *)
  let rec meilleur_score = fun
    | [] _ _-> failwith "J'ai perdu la partie !!!"
    | [a] s o -> let n = (evalue [a] 6 couleur couleur) in if n>=s then a else o
    | (t::q) s o -> let n = (evalue [t] 6 couleur couleur) in
      if n>=s then meilleur_score q n t else meilleur_score q s o
  in let l = transitions_possibles (a,b,c,d,e) couleur in
  let s = meilleur_score l (-20) (a,b,c,d,e) in s ;;

let jeu couleur =
(* Fonction principale du jeu *)
let p = ref (Blanc,Noir,Noir,Blanc,Vide) and l = ref [] and
adversaire = if couleur=Blanc then Noir else Blanc
in
begin
  if couleur = Noir then print_string "Vous avez les Blancs "
  else print_string "Vous avez les noirs ";
  print_newline();
  affiche_position !p;
  while true do
    p := donne_position();
    affiche_position !p;
    p := meilleur_coup !p couleur;
  print_string "Je joue :";
  print_newline();
  affiche_position !p;
  l:=transitions_possibles !p adversaire;
  if (list_length !l)=0 then failwith "Vous avez perdu !";
  done;
end;;

```

3.6.) Jeu de « tic-tac-toe ».

On pourrait implémenter de la même manière un jeu de tic-tac-toe (morpion sur un damier de 3*3) :

```

type Situation = Croix | Rond | Vide;;

let note p couleur =
(* Attribue une note une position *)
let egal a b c = if (p.(a)=p.(b)) && (p.(b)=p.(c)) then true else false in
if egal 0 1 2 then
  if p.(0)=Vide then 0 else if p.(0) = couleur then 10 else (-10)
else
if egal 3 4 5 then
  if p.(3)=Vide then 0 else if p.(3) = couleur then 10 else (-10)
else
if egal 6 7 8 then
  if p.(6)=Vide then 0 else if p.(6) = couleur then 10 else (-10)

```

```

else
if egal 0 3 6 then
  if p.(0)=Vide then 0 else if p.(0) = couleur then 10 else (-10)
else
if egal 1 4 7 then
  if p.(1)=Vide then 0 else if p.(1) = couleur then 10 else (-10)
else
if egal 2 5 8 then
  if p.(2)=Vide then 0 else if p.(2) = couleur then 10 else (-10)
else
if egal 0 4 8 then
  if p.(0)=Vide then 0 else if p.(0) = couleur then 10 else (-10)
else
if egal 6 4 2 then
  if p.(6)=Vide then 0 else if p.(6) = couleur then 10 else (-10)
else 0;;

```

```

let transitions_possibles v couleur =
(* Transitions possibles à partir d'une position *)
let l = ref [] and t = ref v in
if ((note v Rond)<>10) && ((note v Croix )<>(-10)) then
  for i = 0 to 8 do
    t:= copy_vect v;
    if !t.(i)=Vide then
      begin
        !t.(i) <- couleur;
        l:= (!t :: !l);
      end
    done;
  !l;;

```

```

let affiche_position p =
(* Affichage de la position passée en paramètre *)
begin
  print_newline();
  for i = 0 to 2 do
    imprime p.(i);
  done;
  print_newline();
  for i=3 to 5 do
    imprime p.(i);
  done;
  print_newline();
  for i=6 to 8 do
    imprime p.(i);
  done;
  print_newline();
end
where imprime = fonction
| Croix -> print_string " X "
| Rond -> print_string " O "
| _ -> print_string " . " ;;

```

```

let donne_position p couleur =
(* Lecture du coup du joueur humain *)
print_newline();
print_string "Donnez votre jeux (1 chiffre en 1 et 9) :";
let s = read_line() in let a=int_of_string s in
begin
  p.(a-1) <- couleur;
end;;

```

```

let rec evaluate = fun
(* Evaluation d'un coup *)
| [] _ _ _ -> 0
| _ _ _ 0 -> 0
| [p] couleur tour pr when tour=couleur ->
  let t1 = if tour=Croix then Rond else Croix
  in let l=transitions_possibles p t1 in
    (note p tour) + evaluate l couleur t1 (pr-1)
| [p] couleur tour pr when tour<>couleur ->
  let t1 = if tour=Croix then Rond else Croix
  in let l=transitions_possibles p t1 in
    minus_int(note p tour) + evaluate l couleur t1 (pr-1)
| (t::q) couleur tour p when tour=couleur ->
  let t1 = if tour=Croix then Rond else Croix in
    (note t tour) + evaluate [t] couleur t1 (p-1) + evaluate q couleur t1 (p-1)
| (t::q) couleur tour p when tour<>couleur ->
  let t1 = if tour=Croix then Rond else Croix in
    minus_int(note t tour) + evaluate [t] couleur t1 (p-1) + evaluate q couleur
t1 (p-1)
| _ _ _ _ -> failwith "Erreur !";

let meilleur_coup p couleur profondeur =
(* Recherche du meilleur coup possible à partir d'une position *)
  let rec meilleur_score = fun
    | [] _ _ -> failwith "Match nul! "
    | [a] s o -> let n = (evaluate [a] couleur couleur profondeur) in if
n>=s then a else o
    | (t::q) s o -> let n = (evaluate [t] couleur couleur profondeur) in
if n>=s then meilleur_score q n t else meilleur_score q s o
  in let l = transitions_possibles p couleur in
  let s = meilleur_score l (-20000) p in
  if s=p then failwith "J'abandonne" else
  s;;

let jeu couleur niveau =
(* Fonction principale du jeu *)
let p = ref [|Vide;Vide;Vide;Vide;Vide;Vide;Vide;Vide;Vide|]
and l = ref []
and la_note = ref 0
and adversaire = if couleur=Croix then Rond else Croix
in
begin
  if couleur = Croix then print_string "Vous avez les ronds(0) "
  else print_string "Vous avez les croix (X) ";
  print_newline();
  affiche_position !p;
  while true do
    donne_position !p adversaire;
    affiche_position !p;
    la_note := note !p couleur;
    if !la_note = (-10) then failwith " Vous avez gagné ! ";
    p :=meilleur_coup !p couleur niveau;
    print_string "Je joue :";
    print_newline();
    affiche_position !p;
    la_note := note !p adversaire;
    if !la_note = (-10) then failwith " Vous avez perdu ! "
  done;
end;;

```

Voici un exemple du déroulement du jeu :

Commentaires	Déroulement du jeu
Lancement du jeu. On passe en paramètres la couleur de l'ordinateur et le niveau de jeu. Affichage de la position initiale	<pre>#jeu Croix 15;; Vous avez les ronds(0)</pre>
Entrée du coup du joueur.	<pre>Donnez votre jeux (1 chiffre en 1 et 9) :3 . . O</pre>
Coup de l'ordinateur.	<pre>Je joue : . . O . X</pre>
Entrée du coup du joueur.	<pre>Donnez votre jeux (1 chiffre en 1 et 9) :1 O . O . X</pre>
Coup de l'ordinateur	<pre>Je joue : O X O . X</pre>
Entrée du coup du joueur	<pre>Donnez votre jeux (1 chiffre en 1 et 9) :8 O X O . X . . O .</pre>
Coup de l'ordinateur	<pre>Je joue : O X O X X . . O .</pre>
Entrée du coup du joueur	<pre>Donnez votre jeux (1 chiffre en 1 et 9) :7 O X O X X . O O .</pre>
Coup de l'ordinateur	<pre>Je joue : O X O X X X O O .</pre>
Résultat du jeu.	<pre>Exception non rattrapée: Failure " Vous avez perdu ! "</pre>

Et voici une version graphique complète :

```
type Situation = Croix | Rond | Vide;;
#open "graphics";;

let note p couleur =
(* Attribue une note une position *)
let egal a b c = if (p.(a)=p.(b)) && (p.(b)=p.(c)) then true else false in
if egal 0 1 2 then
  if p.(0)=Vide then 0 else if p.(0) = couleur then 10 else (-10)
else
  if egal 3 4 5 then
    if p.(3)=Vide then 0 else if p.(3) = couleur then 10 else (-10)
  else
    if egal 6 7 8 then
      if p.(6)=Vide then 0 else if p.(6) = couleur then 10 else (-10)
    else
      if egal 0 3 6 then
```

```

    if p.(0)=Vide then 0 else if p.(0) = couleur then 10 else (-10)
  else
  if egal 1 4 7 then
    if p.(1)=Vide then 0 else if p.(1) = couleur then 10 else (-10)
  else
  if egal 2 5 8 then
    if p.(2)=Vide then 0 else if p.(2) = couleur then 10 else (-10)
  else
  if egal 0 4 8 then
    if p.(0)=Vide then 0 else if p.(0) = couleur then 10 else (-10)
  else
  if egal 6 4 2 then
    if p.(6)=Vide then 0 else if p.(6) = couleur then 10 else (-10)
  else 0;;

let transitions_possibles v couleur =
(* Transitions possibles à partir d'une position *)
let l = ref [] and t = ref v in
if ((note v Rond)<>10) && ((note v Croix )<>(-10)) then
  for i = 0 to 8 do
    t:= copy_vect v;
    if !t.(i)=Vide then
      begin
        !t.(i) <- couleur;
        l:= (!t :: !l);
      end
    done;
  !l;;

let message s =
  set_color white;fill_rect 1 (size_y()-300) (size_x()) 30;
  set_color black;moveto 10 (size_y()-300);
  draw_string s;;

let affiche_position p =
(* Affichage de la position passée en paramètre *)
begin
  let sy = size_y() in
  begin
    set_color black;
    (* lignes horizontales *)
    moveto ( 100) (sy - 100);
    lineto ( 250) (sy - 100);
    moveto ( 100) (sy - 150);
    lineto ( 250) (sy - 150);
    (* lignes verticales *)
    moveto ( 150) (sy - 50);
    lineto ( 150) (sy - 200);
    moveto ( 200) (sy - 50);
    lineto ( 200) (sy - 200);
  end;
  for i = 0 to 2 do
    imprime p.(i) (i+1) 1;
  done;
  print_newline();
  for i=3 to 5 do
    imprime p.(i) (i-2) 2;
  done;
  print_newline();
  for i=6 to 8 do
    imprime p.(i) (i-5) 3;
  done;
  print_newline();
end
where imprime = fun
  | Croix x y -> set_color black;
  moveto (55+(x*50)) (size_y()-5-(50*y));lineto (95+(x*50)) (size_y()-45-
(50*y));
  moveto (95+(x*50)) (size_y()-5-(50*y));lineto (55+(x*50)) (size_y()-45-
(50*y));
  | Rond x y -> set_color black; draw_circle (75+(x*50)) (size_y()-25-(50*y)) 20
  | _ x y -> set_color white;fill_rect (51+(x*50)) (size_y()-49-(50*y)) 45 45 ;;

```

```

(* Lecture du coup du joueur humain *)
let ok = ref false and i = ref 0 and
    m = if couleur = Rond then "A vous de jouer. (Vous avez les ronds)"
let donne_position p couleur =

    else "A vous de jouer. (Vous avez les croix)" in

message m;
while not !ok do
    let ev = wait_next_event [Button_down] and sy = size_y() in
    let x = ev.mouse_x and y=ev.mouse_y in
    if (x>100) && (x<145) then i:=1
    else
        if (x>155) && (x<195) then i:=2
        else if (x>205) && (x<240) then i:=3 else i:=(-1);
        if !i>0 then if (y<(sy-50)) && (y>(sy-100)) then i:=!i
        else if (y<(sy-100)) && (y>(sy-150)) then i:=!i+3 else
            if (y<sy-150) && (y>(sy-200)) then i:=!i+6 else i:=0;
        if (!i>0) && (!i<10) && p.(!i-1) = Vide then
            begin
                ok:=true;
                p.(!i-1) <- couleur;
            end
        done;;

let rec evalue = fun
(* Evaluation d'un coup *)
| [] _ _ _ -> 0
| _ _ _ 0 -> 0
| [p] couleur tour pr when tour=couleur ->
    let t1 = if tour=Croix then Rond else Croix
    in let l=transitions_possibles p t1 in
    (note p tour) + evalue l couleur t1 (pr-1)
| [p] couleur tour pr when tour<>couleur ->
    let t1 = if tour=Croix then Rond else Croix
    in let l=transitions_possibles p t1 in
    minus_int(note p tour) + evalue l couleur t1 (pr-1)
| (t::q) couleur tour p when tour=couleur ->
    let t1 = if tour=Croix then Rond else Croix in
    (note t tour) + evalue [t] couleur t1 (p-1) + evalue q couleur t1 (p-1)
| (t::q) couleur tour p when tour<>couleur ->
    let t1 = if tour=Croix then Rond else Croix in
    minus_int(note t tour) + evalue [t] couleur t1 (p-1) + evalue q couleur t1 (p-1)
| _ _ _ _ -> failwith "Erreur !";;

let meilleur_coup p couleur profondeur =
(* Recherche du meilleur coup possible à partir d'une position *)
    let rec meilleur_score = fun
        | [] _ _ -> failwith "Match nul! "
        | [a] s o -> let n = (evalue [a] couleur couleur profondeur) in if n>=s then
            a
        else o
        | (t::q) s o -> let n = (evalue [t] couleur couleur profondeur) in
            if n>=s then meilleur_score q n t else meilleur_score q s o
    in let l = transitions_possibles p couleur in
    let s = meilleur_score l (-2000000) p in
    if s=p then failwith "J'abandonne" else
    s;;

let jeu couleur niveau =
(* Fonction principale du jeu *)
let p = ref [|Vide;Vide;Vide;Vide;Vide;Vide;Vide;Vide;Vide|]
and l = ref []
and la_note = ref 0
and adversaire = if couleur=Croix then Rond else Croix
in
begin
    open_graph "dessin";
    clear_graph();
    if couleur = Croix then message "Vous avez les ronds(0) "
    else message "Vous avez les croix (X) ";
    affiche_position !p;
    while true do

```

```

    affiche_position !p;
    la_note := note !p couleur;
    if !la_note = (-10) then
    donne_position !p adversaire;
begin
    message " Vous avez gagné ! ";
    wait_next_event [Button_down;Key_pressed];
    raise Exit;
end
else
begin

```

```

    message "Je réfléchis...";

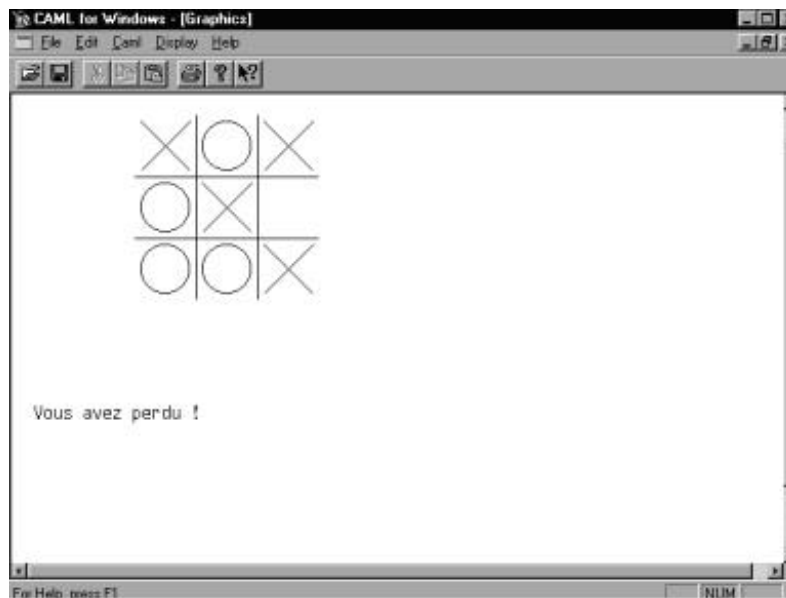
```

```

    p :=meilleur_coup !p couleur niveau;
    affiche_position !p;
    la_note := note !p adversaire;
    if !la_note = (-10) then
    begin
        message " Vous avez perdu ! ";
        wait_next_event [Button_down;Key_pressed];
        raise Exit;
    end;
end;
done;
end;;

```

ce qui donnera :



4.) Programmation des jeux d'échecs.

4.1.) Introduction.

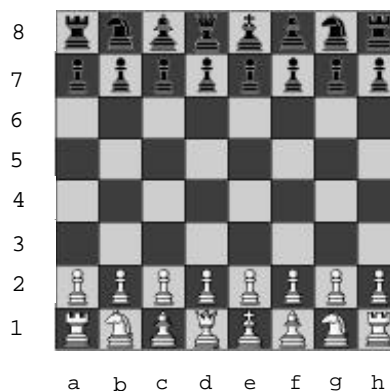


Le jeu d'échecs, est un jeu de stratégie composé d'un plateau de 64 cases, alternativement noires et blanches, sur lequel deux joueurs font manœuvrer deux séries (une noire et une blanche) de seize pièces de différentes valeurs. Chaque série comprend un roi, une dame, deux fous, deux cavaliers, deux tours et huit pions. L'échiquier est disposé entre les joueurs de manière à ce que la case située à la droite de chacun d'eux soit blanche. Les huit lignes verticales situées entre les deux joueurs sont appelées colonnes!; huit lignes horizontales qui les coupent à angle droit sont appelées rangées et les lignes qui traversent l'échiquier en diagonale sont appelées diagonales.

Le but du jeu, inspiré de l'art de la guerre, est de s'emparer (c'est-à-dire de mettre échec et mat) du roi de l'adversaire.

La notation algébrique attribuée à chaque case de l'échiquier une lettre et un nombre : on attribue aux huit colonnes (en commençant à la gauche des blancs) une lettre de l'alphabet (de a à h), et les huit rangées (en commençant du côté des blancs) sont numérotées de 1 à 8. Les figures (et non les pions) sont désignées par leur initiale. Ainsi Df8 signifie que la dame se déplace à l'intersection de la huitième rangée et de la colonne f. La prise d'une pièce est indiquée par un x — comme dans Dxb5 : la dame prend une pièce ou un pion en b5.

Le premier vrai tournoi international, qui se déroula à Londres en 1851, fut remporté par un Allemand, Adolf Anderssen.



Aux échecs, la force d'un joueur s'exprime en points Elo :

Nombre de points Elo	Force du joueur.
1000	Débutants (connaissant les règles)
1550	La moyenne nationale des joueurs français
2000	Un très bon joueur de club
2400	Un Maître international (MI)
2550	Un Grand maître international (GMI)
2800	Le champion du Monde

La plupart des jeux d'échecs vendus dans le commerce (Fritz 5.0, Genius 4.0, Hiarcs, Virtual Chess etc. ...) tournent autour de 2550 Elo pour les meilleurs, mais valent plus lorsqu'ils jouent en parties rapides : 5 min (blitz) chacun par exemple.

4.2.) Représentation des états possibles du jeu.

La première difficulté dans la programmation d'un jeu d'échecs (comme pour tout autre jeu) consiste à choisir une manière efficace de représenter l'état du jeu. En ce qui concerne le jeu d'échecs, Il existe

deux grandes techniques : le codage en 2 dimensions et le codage en 1 dimension.

4.2.1.) CODAGE EN 2 DIMENSIONS :

Cette première méthode, très peu utilisée, code sur une matrice en 2 dimensions de dimension 8*8 ou plus, si vous voulez sauvegarder des informations complémentaires. Cette méthode est très intuitive, car vous faites l'appel à la variable échiquier(ligne, colonne). Cependant les appels aux tableaux à 2 dimensions sont assez longs et de plus pour savoir si vous êtes dans l'échiquier, vous devez faire 4 tests (si la ligne est comprise entre 1 et 8 et le même test pour la colonne).

Dans tous les cas vous balayez l'ensemble des cases, et à chaque fois que vous rencontrez une pièce, vous déterminer l'ensemble des cases où cette pièce peut se déplacer :

3 types de pièces sont à envisager :

1. Les pions (il faut envisager tous les cas : poussée de 2 pions, les prises, les promotions et les prises en passant) : la description de l'algorithme est inintéressante ici car trop fastidieuse.
2. Les pièces à courte portée : le roi et le cavalier.
3. Les pièces à longues portées (Fou, Dame, Tour) qui ont toutes un comportement similaire (il n'y a que le vecteur direction qui change)

1er Cas : Pièce à courte portée : ex cavalier sur la case 2,7 :

Vous regardez toutes les cases possibles avec les vecteurs directions suivants:

$$(2,1),(2,-1),(1,-2),(-1,-2),(-2,-1),(-2,1),(1,2)$$

Ici, cela donne :

- 1^{ère} case : $(2,7) + (2,1) = 4,8$ qui est bien sur l'échiquier (reste à voir si cette case est libre ou occupée par un ennemi)
- 2^{ème} case : $(2,7) + (2,-1) = 4,6$: OK
- 3^{ème} case : $(2,7) + (1,-2) = 4,6$: OK
- 5^{ème} case : $(2,7) + (-2,-1) = (0,6)$: Pas une case de l'échiquier, donc le cavalier ne peut y aller.

1,8	2,8	3,8	4,8	5,8	6,8	7,8	8,8
1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7
1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6
1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5
1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4
1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3
1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2
1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1

2ème cas: pièce de longue portée

Ex : Fou blanc en 6,3 avec Pions noirs en 3,6 et 7,4 (cases en gras) et un roi blanc en 8,1 (en vert). Les vecteurs directions sont pour le fou : (-1,1);(1,1),(1,-1),(-1,-1)

1,8	2,8	3,8	4,8	5,8	6,8	7,8	8,8
1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7
1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6
1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5
1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4
1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3
1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2
1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1

- **1ere direction :** $(6,3) + (-1,1) = (5,4)$: il n'y a pas de pièce, le coup $(6,3) \rightarrow (5,4)$ est valable ; tant qu'il n'y a pas de pièce on ne s'arrête pas, le programme va vérifier si la pièce peut aller plus loin :
 $(5,4) + (-1,1) = (4,5)$: On enregistre le coup $(6,3) \rightarrow (4,5)$ et on continue !
 $(4,5) + (-1,1) = (3,6)$: On enregistre le coup et on s'arrête
- **2ème direction :** $(6,3) + (1,1) = (7,4)$: il y a un pion noir: on peut donc enregistrer le coup, mais on ne poursuit pas dans cette direction
- **3ème direction :** $(6,3) + (1,-1) = (7,2)$: OK, on enregistre le coup $(6,3) \rightarrow (7,2)$ et on continue
 $(7,2) + (1,-1) = (8,1)$: la case est valide mais il y a une pièce de même couleur que le fou (roi blanc), le programme n'enregistre pas le coup $(6,3) \rightarrow (8,1)$.
- **4ème direction :** $(6,3) + (-1,-1) = (5,2)$: OK, on enregistre le coup $(6,3) \rightarrow (5,2)$ et on continue
 $(5,2) + (-1,-1) = (4,1)$: OK, on enregistre le coup $(6,3) \rightarrow (4,1)$ et on continue
 $(4,1) + (-1,-1) = (3,0)$: pas dans l'échiquier donc on s'arrête.

4.2.2.) CODAGE SUR UN TABLEAU A 1 DIMENSION :

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Si vous voulez connaître les mouvements des tours à partir de la case 25, vous n'avez alors qu'à balayer l'échiquier:

- case 17,9,1 et on s'arrête puisque la prochaine case serait $-7 < 0$
- case 33,41,49,57 et on s'arrête puisque la prochaine case serait $65 > 63$
- case 26,27,28,29,30,31, 32 (???) :(cela ne correspond plus à la même ligne, mais ça vous ne pouvez pas le savoir.

L'avantage de la 2ème méthode est la rapidité d'exécution (car l'accès au tableau à 1 dimension est nettement plus rapide que l'accès en 2 dimensions) et la sauvegarde des coups (1 case = 1 entier).

4.3.) Liste des positions accessibles.

Comme dans les exemples de jeux précédents, il faut des procédures pour que l'ordinateur connaisse les coups jouables dans une position, c'est à dire l'ensemble des positions accessibles. Cela revient en fait à apprendre à l'ordinateur les règles du jeu.

- La dame se déplace d'une ou de plusieurs cases, dans n'importe quelle direction, verticalement, horizontalement ou en diagonale. Si elle rencontre une pièce de son camp, elle doit s'arrêter. Si elle rencontre une pièce adverse, elle peut également s'arrêter ou capturer celle-ci (qui est ensuite retirée de l'échiquier) et occuper sa case.
- La tour se déplace verticalement ou horizontalement d'une ou plusieurs cases, dans n'importe quelle direction.
- Le fou se déplace en diagonale d'une ou de plusieurs cases dans n'importe quelle direction.
- Le cavalier est la seule pièce pouvant passer au-dessus d'une autre pièce ou d'un pion quelle que soit sa couleur. Il se déplace en L, c'est-à-dire qu'il se déplace verticalement ou horizontalement de deux cases dans n'importe quelle direction, puis d'une case, perpendiculairement aux deux précédentes. Son déplacement doit se terminer sur une case de couleur différente de celle d'où il est parti. Cette case est occupée ou non par une

pièce adverse (qui, dans ce cas, est alors capturée).

- Un pion ne peut se déplacer que verticalement. Au cours du premier déplacement, il peut avancer d'une ou de deux cases ; ensuite il ne se déplace que d'une seule case à la fois. Il ne peut se déplacer que si la case voisine est inoccupée. Un pion peut capturer n'importe quelle figure ou pion adverse situé devant lui sur une case diagonale.
- Le roi peut se déplacer dans n'importe quelle direction sur toutes les cases adjacentes, à condition que la case qu'il veut atteindre ne soit pas menacée par une pièce adverse ou occupée par une pièce de même couleur. Le roi peut capturer une pièce adverse si celle-ci n'est pas protégée et même si cette pièce le met en échec.

En plus des règles simples de déplacement des pièces, il faut donner les règles sur le roque (incluant la règle que l'on ne peut pas roquer si le roi ou la tour a déjà bougé, si le roi est en échec ou s'il traverse en roquant une case contrôlée par l'adversaire), la prise en passant, la promotion, le mat et le pat. La programmation de ces règles est certes fastidieuse mais elle ne pose aucun problème particulier.

4.4.) Les algorithmes d'évaluation statique de la position.

Une fois qu'on a défini une structure de donnée et une fonction permettant de lister tous les coups possibles à partir d'une position donnée, il ne reste plus qu'à faire « réfléchir » la machine. Pour cela, il faut une fonction qui puisse « noter » une position, c'est à dire attribuer une valeur à chaque position.

L'écriture de cette fonction d'évaluation est la partie la plus délicate, car il faut savoir à la fois assez bien jouer aux échecs, c'est-à-dire connaître la valeur de chaque élément stratégique et être un bon programmeur pour que la durée de l'évaluation d'une position soit la plus courte possible.

L'évaluation est un chiffre compris entre - infini (= Mat Noir) et + infini (=Mat blanc). Dans la pratique ce sera une grande valeur ou une petite valeur.

L'évaluation d'une position se compose de deux éléments : Le dénombrement des pièces et la configuration du jeu.

4.4.1.) DENOMBREMENT

On a l'habitude aux échecs d'avoir comme valeur pour les pièces :

Pièces	Valeur en unité de pion
Pion	1
Cavalier	3
Fou	3
Tour	5
Dame	9,5
Roi	+ infini

Une évaluation égal à -4 signifie que les noirs ont une avance équivalente à 4 pions (ou un cavalier et un pion ...)

4.4.2.) ANALYSE DE LA CONFIGURATION

Il faut aussi tenir compte de facteurs autres que le dénombrement des pièces. Voici quelques éléments de base :

- L'évaluation des tours : on donnera un bonus si celles-ci n'ont pas de pions devant elles (colonnes ouvertes ou semi-ouvertes) (ce qui est facile à vérifier si on code la position avec un tableau), si elles sont sur la 7ème traverse ou la 8ème traverse.
- L'évaluation des fous : bonus s'il y a la paire de fous (facile à vérifier en balayant l'échiquier), s'il n'y a pas trop de pions du même camp qui sont sur leur couleur (sinon il risque de buter contre eux).
- L'évaluation des cavaliers : On prend en compte surtout la position sur l'échiquier : si un cavalier est au centre, son activité est plus importante que s'il est sur le côté (car dans ce cas, il a très peu de cases pour bouger).
- L'évaluation de la dame : pas de règle particulière.
- L'évaluation du roi : en début et milieu de jeu, il faut que le roi soit à l'abri, on donnera un bonus lorsque le roi aura roqué, et un malus si par exemple il a perdu le droit de roquer. On donnera un malus si les pions du roque ont été avancés (ce qui signifie que le roque est détérioré. En finale (nécessité d'avoir une fonction qui détermine dans quelle phase de jeu on est), il faut enlever ces malus, et promouvoir une position du roi active (au centre en général).
- L'évaluation des pions: il faut que les pions ne soient pas doublés, pas isolés (un pion sans soutien de part et d'autre), pas arriéré : si jamais cela leur arrive (il faut évidemment que la fonction d'évaluation le détecte), un malus sera appliqué. Dans le cas où on a des pions au centre, des pions passés, des pions passés protégés, ou des pions proches de la promotion, on appliquera dans ce cas-là un bonus.

4.5.) Utilisation du MiniMax.

L'exploration des coups se fera à l'aide de l'algorithme du MiniMax, comme pour un simple jeu de tic-tac-toe. La difficulté, aux échecs, réside dans le nombre de combinaisons possibles à chaque coup. En effet, il y a en moyenne 35 coups possibles dans une position de milieu de jeu : or si on analyse 35 coups blancs, et qu'à chaque coup blanc, il y a 35 réponses possibles et qu'à chacun de ces coups noirs 35 réponses possibles... Cela fait $35 \times 35 \times \dots$ positions à calculer. Pour calculer à une profondeur de 6 demi-coups (un demi-coup est un coup blanc ou un coup noir) - ce qui est très faible - il faudrait théoriquement calculer 35^6 positions = 1838265625 positions ! Et même si le joueur ne sélectionne que 3 coups sur 35, il devra calculer 729 positions ce qui est énorme si on considère le temps moyen de 3 minutes par coups en compétition. Il faudra donc utiliser des techniques qui permettront de réduire le nombre de coups à analyser et c'est en cela que la programmation des jeux d'échecs est délicate.

4.5.1.) PROFONDEUR 1

Ici, on se contente de rechercher le meilleur coup possible des blancs. L'utilisation du MiniMax à une profondeur 1 se fera en 4 étapes :

1. Faire la liste de tous les coups blancs possibles.
2. Pour chaque coup possible, appeler la fonction d'évaluation.
3. Retenir le coup qui obtient la meilleure note.
4. Jouer ce coup.

4.5.2.) PROFONDEUR 2

Maintenant, on analyse tous les coups blancs et, pour chaque coup, les réponses noires possibles.

1. Faire la liste de tous les coups blancs possibles.
2. Pour chaque coup possible, appeler la fonction d'évaluation et conserver la note B. A partir de cette position, lister tous les coups noirs possibles et retenir le coup qui génère la note la plus basse (pour les blancs). Soustraire à B ce coup noir.
3. Retenir le coup qui obtient la meilleure note.
4. Jouer ce coup.

Evaluer une position à la profondeur 2, ressemble donc à l'analyse profondeur 1, mais au lieu d'appeler la fonction d'évaluation statique, vous appellerez la fonction de recherche à la profondeur 1. Et ainsi de suite, calculer à la profondeur n, c'est calculer avec la fonction d'évaluation profondeur 1, mais avec une évaluation à la profondeur n-1.

C'est le principe même de la récursivité: Votre fonction va s'appeler avec le paramètre (profondeur-1) jusqu'à ce que profondeur=0, et dans ce cas elle appellera la fonction Évaluation toute simple.

Si on résume cet algorithme (minimax : qui vient de maximiser et minimiser), on se rend compte qu'à chaque niveau, le joueur qui a le trait va essayer de maximiser son jeu (=jouer le meilleur coup) et minimiser le jeu de son adversaire. On considère évidemment ici que chaque joueur joue le meilleur coup (on ne s'attend pas à une bourde de l'adversaire).

Une des caractéristiques de l'algorithme Minimax est sa croissance exponentielle : si on calcule au niveau 3, cela fait $35*35*35$ coups à analyser, ce qui commence à faire beaucoup, et je ne vous parle pas à d'un niveau 15.

4.5.3.) LIMITES DU MINIMAX

Avec MinMax, le programme aura un problème : **l'effet d'horizon**: il s'arrêtera à un niveau n, mais s'il y a un coup noir qui gagne une dame au niveau n+1, il ne le verra pas: il faut donc dans certaines circonstances prolonger d'un niveau (lors des prises, des échecs, des promotions). En effet sinon soit il jouera un coup et se fera prendre (en général lorsque la profondeur d'analyse est impair : il ne voit pas le dernier coup de l'adversaire; et lorsque la profondeur est paire, il a "peur" de l'ennemi car il croit toujours que celui-ci peut lui prendre des pièces car il ne voit pas que ses pièces sont protégées) Donnons 2 exemples à une profondeur 1 (impaire) et une profondeur 2 (paire) mais qui peuvent se généraliser à un niveau n, et qui permettent d'illustrer notre propos:

Profondeur = 1 : Imaginer une position avec une dame au centre de l'échiquier et qui peut prendre plein de pièces ennemies mais qui sont protégées: l'ordinateur va calculer que la prise d'une pièce est intéressante (en apparence à une prof=1) et il jouera ce coup même si la dame est prise le coup suivant et tout cela à cause de l'effet d'horizon.

Profondeur = 2: imaginez une position quelconque dans lesquels les pièces blanches sont protégées et une pièce noire (dame) est au centre: les blancs ont le trait: ils vont calculer que s'ils jouent un coup, la dame noire va leur prendre une pièce (ce qu'ils ne voient pas, c'est que les blancs reprennent juste après et que la prise d'une pièce blanche par la dame noire est une énorme erreur). Ils vont donc rapatrier leurs pièces pour que la dame "ne puisse pas leur prendre leur pièce".

Profondeur=n : comme il faut bien arrêter le calcul à une profondeur donnée, l'effet d'horizon pose toujours problème

Prenons un autre exemple qui arrive très fréquemment:

Supposons qu'il existe une combinaison qui conduise inévitablement à la perte de la dame. Supposons de plus que le camp qui va perdre la reine puisse retarder cette perte en sacrifiant d'autres pièces moins importantes à un autre endroit de l'échiquier. Comme la perte de la reine est inéluctable, il est absurde de continuer à jouer des coups de retardement qui sacrifient encore plus de pièces. Mais les coups d'attente peuvent mener le jeu au-delà de la profondeur limite décidée pour la recherche, ce qui

a pour effet de masquer les catastrophes.

Le remède à ce genre d'inconvénient est de vérifier que lorsqu'un joueur prend une pièce avec une de ses pièces, cette dernière ne soit pas reprise auquel cas, il faudra calculer la perte ou le gain matériel. Pour éviter maintenant de se faire prendre des pièces, il faut vérifier que le camp adverse ne possède pas de possibilité pour prendre ses propres pièces. Certains programmeurs ont trouvé des fonctions qui déterminent s'il faut aller plus loin dans la recherche ou pas: ils vérifient que la position est stable, c'est-à-dire que même si un joueur joue un coup, il ne peut pas attaquer, ou créer un déséquilibre. Il utilise l'algorithme null move heuristic : ils vérifient que même si on passait son tour et qu'on laissait donc l'adversaire jouer 2 coups de suites, l'adversaire n'arriverait pas à prendre grand chose dans votre camp: dans ce cas, on peut considérer que votre position est stable.

Toutefois, l'effet d'horizon n'est résolu que dans certains cas (prises, promotions, échec etc.), mais pas dans tous, sinon il faudrait un temps fou de calcul pour l'ordinateur: il faut donc arriver à ne pas pousser trop loin les calculs, mais en s'assurant tout de même que la position est stable (pas de prises de pièces). En général, on poursuit les calculs pour les prises, les promotions; les échecs au roi (ce qui permet de déceler des mat même s'ils sont en plusieurs coups. Pour l'instant, mon jeu ne prolonge qu'en cas de prise et il prolonge très peu sinon, il calcule trop longtemps. La difficulté est donc de trouver un compromis entre précision due au calcul exhaustif et la contrainte liée au temps de réflexion qui empêche de calculer très loin.

Il est toujours très difficile de savoir s'il faut prolonger d'un demi-coup la recherche sur tout l'arbre ou s'il faut regarder ponctuellement plus loin pour essayer d'éviter l'effet d'horizon.

4.5.4.) AUTRES ALGORITHMES

-Killer Move heuristic il s'agit du même genre d'idée que les tris de tableaux : en effet, les coups qui ont déclenché une coupure d'arbre sont sensés être des coups forts, dans une autre variante, ils ont une grande chance d'être également très forts. On sauvegarde 2 ou 3 coups qui ont déclenché cette coupure pour chacun des niveaux (1 à n) et on les place systématiquement en premier dans les listes s'ils existent : Les nouveaux coups qui provoquent une coupure remplaceront les anciens.

Ex : si dans une variante un coup comme Cxe5 provoque des coupures, on va le placer s'il existe dans d'autres variantes en tête, il risque de provoquer une coupure : si ça ne marche pas, il n'y aura presque pas de temps de perdu et si ça marche, cela économise beaucoup du temps.

-Aspiration Search: il s'agit de réduire la fenêtre de l'algorithme Alpha non pas à une fenêtre - infini (alpha) et + infini (bêta) mais avec $\alpha := \text{valeur} - E$ et $\beta = \text{valeur} + E$ avec E un entier qu'il faut adapter (il paraîtrait que le mieux est entre 1 et 1/3 de la valeur du pion) et Valeur est l'évaluation que l'ordinateur a la plus grande chance de trouver. Ce genre d'algorithme permettrait de gagner un facteur 4 en vitesse, mais je ne comprends pas encore bien comment : en effet, les coups qui ne rentrent pas dans la fenêtre (si l'évaluation n'est pas comprise entre valeur+E et valeur-E) sont recalculées mais différemment. Si quelqu'un a des informations plus précises sur cet algorithme, cela m'intéresse.

-Principal Variation Search permet de calculer plus profondément la variante principale, c'est-à-dire la variante dont l'évaluation est nettement au dessus des autres, donc celle qui doit normalement être joué, sauf si une recherche plus approfondie montre que la variante est mauvaise.

Transposition Table : il s'agit d'enregistrer les positions déjà calculées afin de ne pas les calculer 2 fois si elles se représentent (utilisation de hash tables). Mais j'ai contourné le problème avec mes listes chaînées qui me semble plus rapide à gérer et aussi puissantes.

Selective Extensions : il s'agit de prolonger les variantes qui sont importantes et/ou forcées, les échecs, les promotions, les prises.

Quiescence search : Au lieu d'appeler la fonction d'évaluation statique (profondeur 0), certains appellent une fonction qu'ils nomment quiescence : celle-ci ne calcule que les captures, et vérifie si la position est stable (si votre fonction d'évaluation vous retourne 0 (égalité) et que vous perdez votre dame juste après, votre évaluation ne sera pas correct, et vous risquez de jouer une mauvaise variante.

Null move heuristic : Il s'agit pour le programme de savoir si une position est stable : vous faites comme si vous passiez votre tour (ce qui n'est pas possible normalement aux échecs) : si votre évaluation ne bouge pas, cela signifie que même si votre adversaire jouait 2 coups de suites, vous ne seriez pas mis en danger, donc votre position est forte et stable, ce qui peut vous dispenser de chercher plus loin. Cependant cet algorithme est très dangereux dans les cas de zugzwang (aucun coup que vous jouez n'est bon, mais vous gagneriez à passer votre tour), car cet algorithme va donner une bonne évaluation, alors que le fait que vous deviez jouer vous handicape sérieusement : C'est pour cette raison, que certains recommandent de le désactiver lorsque le nombre de pièces est faible.

5.) Exercices.

5.1.) Exercices.

EXERCICE 9.1.

On considère le jeu suivant : Au départ on dispose quelques allumettes sur une table (au minimum 10). Le premier joueur prend 1 allumette. Chaque joueur suivant peut choisir de prendre entre 1 allumette et le double du choix du joueur précédent. Par exemple, si le joueur précédent prend 2 allumettes, on pourra prendre 1,2,3 ou 4 allumettes. Le perdant est celui qui retire la dernière allumette.

Programmez ce jeu en Caml. (Uniquement la fonction d'évaluation d'un coup).

EXERCICE 9.2

Calculez e , la base des logarithmes naturels, en utilisant le développement suivant :

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} \dots + \frac{1}{n!}$$

EXERCICE 9.3

Lorsqu'on élève au carré un nombre de Kaprekar à n chiffres et qu'on ajoute les n chiffres de droite aux n ou aux $n-1$ chiffres de gauche, on retrouve le nombre d'origine.

Exemples :

Nombre de Kaprekar à 1 chiffre :	$9^2 = 81$	$8 + 1 = 9$
Nombre de Kaprekar à 2 chiffres :	$55^2 = 3025$	$30+25 = 55$
Nombre de Kaprekar à 3 chiffres :	$297^2 = 88209$	$88+209 = 297$

Trouvez les 10 premiers nombres de Kaprekar.

EXERCICE 9.4

La fonction suivante est écrite en SQR (Structured Query Report language). Elle calcule la clé d'un numéro d'établissement de santé (hôpital, clinique...) ou d'un numéro de professionnel de santé (médecin, pharmacien, ambulancier...). Ecrivez son équivalent en Caml.

```

BEGIN-PROCEDURE calc_cle_12($num_dpt, $num_cat, $num_ord, :#total)
  LET $nombre = $num_dpt || $num_cat || $num_ord
  LET #passe = 1
  LET #total = 0
  WHILE 1
    IF #passe > 8
      BREAK
    END-IF
    LET $c = substr($nombre,#passe,1)
    IF $c = 'A'
      LET #n1 = 10
    ELSE
      IF $c = 'B'
        LET #n1 = 11
      ELSE
        LET #n1 = to_number($c)
      END-IF
    END-IF
    IF (#passe=2) OR (#passe=4) OR (#passe=6) OR (#passe=8)
      LET #n1 = #n1 * 2
    END-IF
    IF #n1 > 9
      LET #total = #total + 1
      LET #n1 = #n1 - 10
    END-IF
    LET #total = #total + #n1
    LET #passe = #passe + 1
  END-WHILE
  LET #total = mod(#total,10)
  LET #total = trunc(10 - #total,0)
  IF #total > 9
    LET #total = 0
  END-IF
END-PROCEDURE

```

- Dans ce langage, le premier caractère devant un nom de variable indique son type : \$ pour une chaîne de caractères, # pour un type numérique (entier et flottants sont confondus).
- Les doubles points (:) placés devant un paramètre, ils indiquent que celui-ci sera retourné par la procédure.
- Le mot clé LET permet d'affecter une valeur à une variable.
- L'opérateur || est l'opérateur de concaténation de chaînes. Il correspond à ^ en Caml.
- L'instruction « WHILE 1... » correspond à « while true do... » en Caml.
- La fonction « substr(\$chaîne,#debut,#longueur) » correspond à « sub_string(chaîne,debut,longueur) » en Caml.
- La fonction « to_number(\$c) » correspond à « int_of_char c » en Caml.
- La fonction « mod(#total,10) » correspond à « total mod 10 » en Caml.
- La fonction « trunc(10 - #total,0) » enlève la partie décimale de #total.
- L'instruction « BREAK » permet de quitter la boucle.

Le tableau suivant donne quelques exemples de numéros établissement avec leurs clés :

Numéro	Clé
25000025	4
25000035	3
06080131	3
93006017	3
49054302	2
57001565	1