L'intelligence **artificielle** des **jeux** de stratégie **classiques**

Introduction à l'IA des jeux de stratégie - Un jeu de stratégie

Beaucoup de jeux traditionnels partagent des caractéristiques intéressantes qui en font une cible "facile" pour le raisonnement mathématique informatisé, et donc l'intelligence artificielle. Les caractéristiques importantes de ces **jeux de stratégie** sont les suivantes :

- Les joueurs jouent à tour de rôle ;
- À chaque tour, le joueur choisit une action parmi un nombre fini (et petit) d'actions valides ;
- On peut prédire l'effet d'une action sur l'état du jeu.

Voici quelques exemples de jeux qui possèdent les caractéristiques voulues :

- Le jeu d'échecs est l'exemple qui va nous accompagner pendant tout cet article. A chaque tour du jeu, le joueur doit bouger une de ses 16 pièces (au plus). Comme les mouvements sont contraints, il y a finalement assez peu de possibilité d'action, environ 35 en moyenne ;
- Le jeu de dames et ses variantes (les dames anglaises par exemple) sont tout à fait similaires au jeu d'échecs, de même qu'Othello. Bien entendu, les règles sont différentes, ce qui change le nombre de coups possibles, mais on reste dans le cadre défini au-dessus ;
- Le backgammon, malgré une composante aléatoire, entre parfaitement dans le cadre des jeux de logique. En effet, en considérant chaque coup du joueur combiné à l'ensemble des lancers de dés possibles, on conserve un nombre relativement raisonnable d'actions possibles, chacune d'elle ayant un effet précis sur l'état du jeu. On pourra donc raisonner au backgammon dans des conditions assez similaires à celles des échecs, en tenant compte bien sûr de l'aspect incertain des actions.

Le but poursuivi tout au long de cet article est de présenter les méthodes mises en œuvre pour construire l'intelligence artificielle d'un jeu de stratégie classique, en l'occurrence les échecs, et de voir comment ces méthodes peuvent s'appliquer également à d'autres jeux de la même famille, comme Othello, le jeu de dames, etc. Nous présenterons de façon générale les algorithmes proposés et nous les illustrerons dans le cas des échecs, en allant dans la plupart des cas jusqu'au code C. Cet article est donc la suite de celui concernant la création d'un jeu d'échecs. Il poursuit en parallèle l'exploration des méthodes de l'intelligence artificielle, but de la rubrique IA de votre magazine préféré.

Pour fixer les idées, voici au contraire quelques exemples de jeux qui ne possèdent pas les caractéristiques voulues :

- L'exemple le plus évident est celui des jeux de stratégie **temps réel**. On peut bien entendu évoquer l'aspect temps réel pour conclure rapidement, mais ce n'est pas le point le plus crucial : l'ordinateur raisonne tellement vite que le temps réel humain n'est rien d'autre que du tour par tour ultra-rapide. La difficulté est en effet issue avant tout de la complexité de ces jeux, en termes d'actions possibles. Le joueur contrôle de très nombreuses unités (40 en moyenne pour un jeu comme Warcraft III), possédant chacune de très nombreuses possibilités d'action (attaque, défense, construction de bâtiments, lancement d'un sort parmi 5 ou 6 possibilités, etc.). Les mouvements autorisés sont en général très riches, et c'est d'ailleurs cette composante et le micro-management qu'elle implique qui font la joie des joueurs. De plus, l'utilisation de formation d'unités complique encore la donne. Enfin, l'effet des actions comporte en général une part importante d'aléa. Tous ces éléments concourent à augmenter de façon radicale le nombre d'actions possibles, à tel point qu'il devient presque infini, au moins du point de vue informatique !
- On retrouve ce problème dans les jeux "du royaume" comme Freeciv [1]. La disparition de la composante temps réel (ce sont des jeux organisés en tours) ne change rien au nombre gigantesque d'actions possibles par tour, à la fois en termes de mouvement d'unités, mais aussi en termes de gestion (construction de bâtiments, répartition des efforts économiques, etc.).
- Citons enfin les jeux d'action à composante stratégique comme *Counter Strike*. Un tel jeu cumule les difficultés : temps réel, actions possibles très nombreuses, effets

aléatoires, auxquels on peut ajouter la dextérité des joueurs humains, qui introduit une nouvelle source de hasard.

En fait, le terme "jeu de stratégie" est très mal choisi car il est clair que les trois catégories de jeux que nous venons de décrire sont éminemment stratégiques ! Il s'agit donc plus d'une dénomination historique (qui recouvre d'ailleurs des jeux assez modernes comme les *wargames*) que d'une véritable définition.

Comment raisonner sur un jeu de stratégie ?

Les caractéristiques particulières des jeux de stratégie permettent un raisonnement systématique pour choisir le meilleur coup à jouer. Il est d'ailleurs très classique de mettre en œuvre un tel raisonnement quand on affronte un adversaire humain : il s'agit de "prédire" l'avenir. Pour ce faire, on considère les coups envisageables dans l'état actuel du jeu, comme par exemple déplacer la reine de telle case à telle autre, etc. Pour chaque coup possible, on calcule mentalement l'état du jeu après son exécution et on répète récursivement le raisonnement du point de vue de l'adversaire. On fait ainsi évoluer l'état du jeu en fonction d'un certain nombre de coups fictifs. Il reste alors à choisir l'état final qui nous plaît le plus (par exemple, celui dans lequel on gagne !) et à revenir en arrière pour choisir le premier coup qui mène à cet état final. Bien entendu, on doit tenir compte de l'intelligence de l'adversaire qui va au contraire tenter de faire évoluer le jeu vers sa propre victoire...

Pour mieux comprendre ce raisonnement, appliquons-le à l'un des jeux les plus simples, à savoir le morpion. Le plateau de jeu est constitué de 9 cases organisées en tableau 3x3. A chaque tour, le joueur place une marque (une croix pour le joueur A et un cercle pour le joueur B) dans une case vide. Le premier joueur qui aligne trois de ses marques a gagné. Considérons par exemple la situation de la **figure 1**.

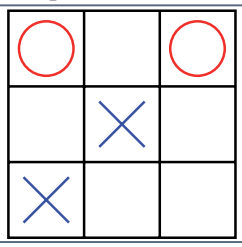
Fig. 1

Le joueur A doit maintenant jouer. Or, on constate que s'il ne joue pas dans la case située au centre de la ligne supérieure, le joueur B (supposé intelligent) pourra y placer un cercle et donc gagner la partie. Pour arriver à cette conclusion, le joueur A considère tous les coups possibles, par exemple celui qui mène à la **figure 2**.

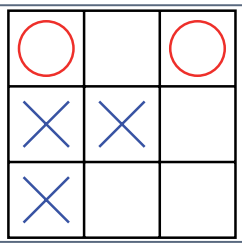
Fig. 2

Il considère ensuite tous les coups possibles pour le joueur B et constate que parmi ceux-ci, celui qui mène à la **figure 3** permet au joueur B de gagner.

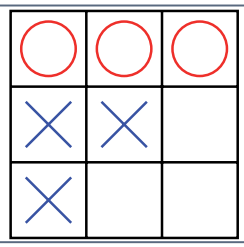
Fig. 3

Comme le joueur B est supposé intelligent, le joueur A ne peut donc pas prendre le risque de jouer le coup qui mène à la **figure 2**. Il constate en explorant tous les coups que seul celui qui mène à la **figure 4** lui permet d'éviter une défaite et il le joue donc.

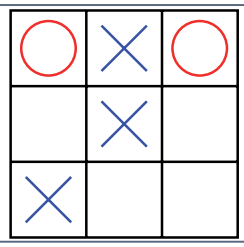
Fig. 4

Le joueur B se lance alors dans le même type d'exploration qui le conduit à jouer au milieu de la ligne du bas, pour éviter une victoire du joueur A.

L'arbre d'un jeu de stratégie

Pour systématiser le mode de raisonnement proposé dans la section précédente, on introduit la notion d'arbre d'un jeu de stratégie. L'idée est de placer à la racine de l'arbre l'état initial du jeu, et d'associer à chaque nœud de l'arbre un état du jeu. Les enfants d'un nœud sont alors les différents états accessibles depuis l'état représenté par le nœud grâce à une action possible pour le joueur concerné. La **figure 5** donne l'exemple du nœud étudié dans la section précédente et de ses 5 fils correspondant aux 5 actions possibles pour le joueur.

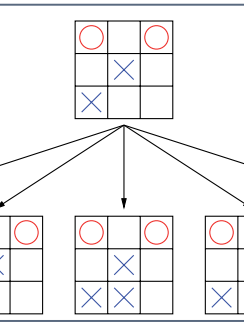
Fig. 5

Pour choisir son prochain coup, le joueur explore l'arbre du jeu à partir de l'état actuel de ce dernier, en essayant d'attribuer un intérêt à chaque coup possible. En général, pour ce faire, il faut continuer l'exploration de façon assez profonde, afin d'atteindre une victoire de l'un ou l'autre des joueurs. Dans l'exemple de la **figure 5**, on peut éliminer

les 4 fils de droite car ils possèdent tous un fils qui correspond à la victoire du joueur B (cf la **figure 6**). Pour ce faire, il faut tout de même explorer tous les fils de chacun de ces fils, soit un total de 16 configurations.

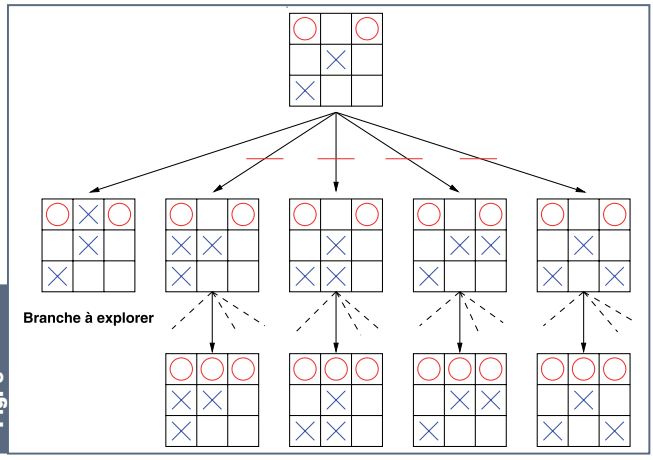

Fig. 6

On constate que les jeux de stratégie, tels que nous les avons définis plus haut, possèdent tous un arbre de déroulement (plus ou moins volumineux) car ils fonctionnent au tour par tour, qu'il existe un nombre fini d'actions possibles à chaque tour et qu'on peut prédire l'effet de chaque action. L'aspect tour par tour correspond aux niveaux de l'arbre, chaque niveau représentant un tour de jeu. Le nombre fini d'actions permet de définir les fils de chaque nœud (et de conserver une taille "raisonnable" à l'arbre) à condition de prédire l'effet d'un coup sur l'état du jeu.

Le but de l'IA d'un jeu de stratégie est donc "simplement" de parcourir efficacement l'arbre du jeu afin de déterminer l'opportunité de jouer telle ou telle action au prochain tour. Nous allons voir dans la section suivante qu'il existe un algorithme récursif simple pour déterminer la qualité d'un coup, puis nous nous intéresserons à des techniques permettant de l'appliquer à des jeux plus complexes que le morpion (et possédant donc un arbre de jeu beaucoup plus grand).

L'algorithme du Minimax

Explication théorique

L'algorithme du Minimax est à la base du travail que fait l'ordinateur pour déterminer le coup qu'il va jouer. Il s'agit donc de construire un arbre exhaustif des coups possibles à jouer par l'ordinateur et par le joueur humain. Comme nous l'avons dit plus haut, chaque niveau de l'arbre correspond aux coups possibles d'un camp donné, alors qu'un nœud correspond à l'état du jeu, par exemple les positions des pièces sur l'échiquier. La construction de l'arbre se fait de manière récursive, nous détaillerons la manière de s'y prendre dans le paragraphe sur l'implémentation.

L'idée principale du Minimax est d'attribuer à chaque nœud une note ou un score qui détermine la qualité de la configuration du jeu pour l'ordinateur (et par symétrie pour le joueur). A priori, les seuls nœuds pour lesquels le score est facile à déterminer sont les feuilles de l'arbre, c'est-à-dire les configurations pour lesquelles le jeu est terminé. En général, les jeux de stratégie se terminent de trois façons : victoire du joueur A, victoire du joueur B ou match nul. On choisit une fois pour toute de prendre arbitrairement le point de vue du joueur A. On attribue ainsi un score positif à une victoire du joueur A, un score négatif à une victoire du joueur B et un score nul à une égalité. Comme les jeux de stratégie sont équilibrés, on évalue symétriquement les scores, c'est-à-dire que la victoire de B donne comme score l'opposé de la victoire de A. Considérons l'exemple du morpion. On peut attribuer le score de 1 à une victoire de A et donc, par symétrie, le score de -1 à une victoire de B. Pour le jeu d'échecs, nous travaillerons d'une façon beaucoup plus subtile, pour des raisons d'efficacité.

L'algorithme du Minimax décrit comment on passe d'un score sur les feuilles à un score sur les nœuds. Il suffit en effet de faire remonter les scores dans l'arbre en considérant toujours le meilleur coup, quel que soit le joueur concerné par le niveau du nœud. C'est de ce principe que provient le nom Minimax. En effet, pour déterminer le score d'un nœud pour lequel le joueur A doit jouer, on choisira toujours le coup qui mène au nœud de score maximal (MAX). Au contraire, pour déterminer le score d'un nœud pour lequel le joueur B doit jouer, on choisira toujours le coup qui mène au nœud de score minimal (MINI) car le score de B est compté négativement.

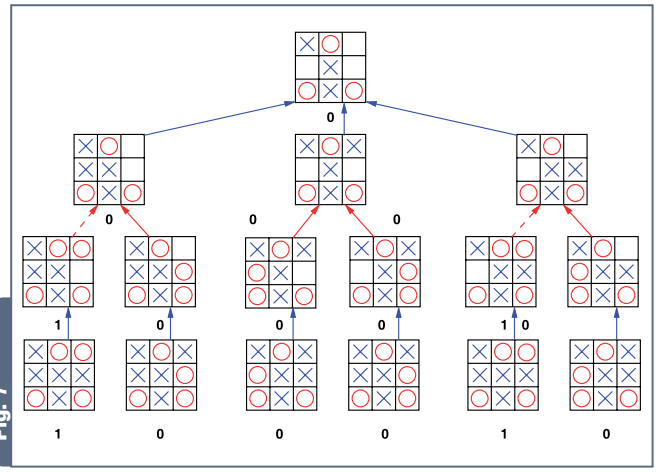
Fig. 7

Voyons un exemple avec le morpion pour bien fixer les idées. La **figure 7** représente l'arbre complet du jeu à partir d'une position intermédiaire. Le joueur A (les croix) doit jouer. Pour ce faire, il calcule l'arbre complet jusqu'à atteindre les feuilles. Dans 2 cas sur 6, les feuilles correspondent à une victoire et ont donc un score de 1. Dans les autres cas, on obtient un match nul, soit un score de 0. Comme le dernier coup du morpion est unique, la phase MAX est réduite à sa plus simple expression et attribue donc aux nœuds situés juste au-dessus des feuilles le même score que ces dernières. Par contre, pour atteindre

les nœuds en question, le joueur B avait toujours deux choix, et on applique ici la phase MINI. Cela signifie en pratique que le joueur B va éviter de jouer un coup qui donne la victoire à son adversaire. Comme sur l'exemple proposé, c'est toujours possible, on obtient finalement un score de 0 pour chaque nœud de la deuxième ligne de la figure. Les flèches en tirets correspondent aux coups que le joueur B ne jouera pas pour éviter d'avantager son adversaire. Enfin, le score de la position initiale est obtenu par une phase MAX qui est encore une fois basique, car les trois coups possibles mènent à des configurations de score identiques. On remarque au passage que les trois coups deviennent donc équivalents alors que celui du milieu mène toujours à un match nul, contrairement aux deux autres qui peuvent conduire à une victoire, en cas d'erreur de l'adversaire. Un des défauts de Minimax est justement ce type de comportement qui consiste à supposer que l'adversaire ne se trompe jamais, ce qui donne un style de jeu assez particulier aux ordinateurs.

Limitation de la profondeur

Le problème de l'algorithme Minimax tel qu'on vient de le présenter est qu'il est monstrueusement coûteux. En effet, on parle bien de développer **tout** l'arbre. Pour prendre l'exemple basique du morpion, on doit donc considérer un arbre de profondeur maximale 9. Au premier coup, on peut choisir entre 9 possibilités, puis entre 8 au deuxième coup, etc. Un raisonnement combinatoire basique permet de majorer le nombre de nœuds par 1 million. Le nombre exact est plus petit car certaines suites de coups conduisent à une victoire rapide, mais le majorant est assez représentatif car il indique bien que l'arbre est assez gros, même pour un jeu très simple. Pour les échecs par exemple, l'arbre est énorme car on compte en moyenne 35 coups possibles par niveau, et 40 niveaux pour une partie (certaines parties peuvent durer beaucoup plus). On doit potentiellement explorer un arbre possédant 35^40 feuilles. Pour fixer les idées, ce nombre est approximativement égal à 2^205, alors qu'un giga-octets correspond à 2^30...

Il est donc indispensable de limiter la profondeur de l'arbre étudié. Celle-ci devra être paire, c'est-à-dire qu'à un coup joué par l'ordinateur, on envisagera toujours la riposte de l'adversaire. Le problème est alors d'évaluer l'état du jeu à un nœud qui n'est pas une feuille. Pour ce faire, on introduit une **heuristique d'évaluation** qui tient compte de différents paramètres propres au jeu lui-même. Nous reviendrons en détail sur ce point dans la suite du texte.

Dans le cas de notre mise en œuvre pratique, le jeu d'échecs, nous évaluerons positivement la position des blancs et celle des noirs, puis nous calculerons le score de l'état du jeu en ôtant au score des blancs celui des noirs. Par conséquent, une configuration sera d'autant plus favorable aux blancs que le résultat de cette différence sera grande. Inversement, il sera d'autant plus favorable aux noirs que le résultat sera petit.

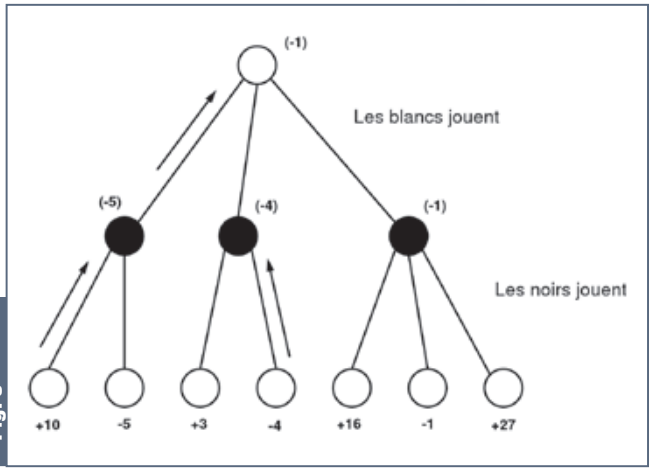

Fig. 8

Considérons un exemple simple dans lequel on explore seulement 2 niveaux. Prenons le cas où l'ordinateur joue avec les blancs (cf **Figure 8**). On considère les coups possibles de l'ordinateur (une moyenne de 35 par tour), puis pour chaque nouvelle configuration de l'échiquier, on en crée de nouvelles, correspondant aux coups joués par les noirs. On évalue ensuite ces différents états de l'échiquier. En partant de la gauche, on va choisir le meilleur coup que peuvent jouer les noirs. Suivant la convention que nous nous sommes fixée, on prend ici le plus petit score : -5. On fait de même sur les nœuds frères. On obtient donc -4, et -1. Maintenant, on remonte d'un niveau, c'est aux blancs de jouer, on va donc choisir le plus gros score : -1. On vient donc de trouver le coup que doivent jouer les blancs pour minimiser la riposte des noirs.

Implémentation en C pour le jeu d'échecs

La mise en place de cet algorithmique se fait naturellement de manière récursive. Le parcours de l'arbre des différentes configurations de l'échiquier sera pseudo-infixe. La première évaluation se fera donc après avoir visité tous les nœuds de la branche de gauche de l'arbre. Concrètement, dans le cas de la figure 8, on commence par longer l'arbre suivant la branche de gauche, on effectue la première évaluation une fois qu'on est arrivé à la feuille de gauche. Le résultat est +10. On remonte ensuite dans l'arbre en renvoyant la valeur +10 qui sera mise à jour si besoin est. On visite ensuite le nœud fils suivant, et ainsi de suite.

Pour initialiser le code de mise à jour, il faudra fixer au départ le score des blancs à -infini et celui des noirs à +infini (l'infini représente dans notre cas une valeur telle que tous les scores possibles soient négligeables face à elle).

Voyons maintenant le code en C, qui permettra de mieux comprendre la mise en place de tous ces mécanismes.

Voici le fichier `ia.c` dont la fonction récursive `explore()` a été modifiée pour ne laisser voir que l'implémentation du `minimax`. Nous verrons plus loin la version implémentant l'alpha/beta. Cette fonction constitue le cœur de l'IA. Elle s'appuie sur la fonction `list_all_mvt()` (définie dans `ARBITRE`) que nous détaillerons juste après. Elle renvoie un pointeur sur une pile de tous les mouvements possibles pour une

couleur donnée. La fonction `check_chessboard()` sera définie dans la section suivante, il s'agit de l'implémentation de l'heuristique d'évaluation d'un nœud. Quant à `move_chessman()`, il faudra se référer à l'article précédent de la série sur l'implémentation du jeu complet. En ce qui concerne le package de pile `STACK`, nous verrons son implémentation dans le prochain article de cette série.

```
/* ia.c */

#include <stdlib.h>
#include "stack.h"
#include "objets.h"
#include "arbitre.h"
#include "ia.h"

int prof_max = 0;
int prof_start = 0;

// la fonction du minimax est appelée dans explore()
int minimax(int score_courant, int best_score, couleur_t couleur)
{
  if(couleur == noir){
    if(score_courant < best_score)
      return(1);
    else
      return(0);
  }else{
    if(score_courant > best_score)
      return(1);
    else
      return(0);
  }
}

int explore(int prof)
{
  extern piece_t *tab[8][8];
  extern coord_t tab_pieces[noir+1][roi+1];
  extern dep_t best_dep;

  // cette pile nous sert à stocker les coups possibles
  stack_t stk;
  dep_t *dep;
  int best_score;
  int score;
  couleur_t couleur;
  piece_t save_end;
  coord_t save_coord;

  // choix de la couleur en fonction de la profondeur
  if(prof%2 == 0){
    couleur = blanc;
    best_score = -100000;
  }else{
    couleur = noir;
    best_score = 100000;
  }

  // fin de l'exploration
  if(prof == prof_max){
    score = check_chessboard();
    return(score);
  }

  // à un niveau donné on liste tous les coups possibles de la couleur concernée
  stk = create_stack(copy_dep);
  list_all_mvt(couleur,stk);

  // tant que la pile des coups n'est pas vide on joue le coup dépilé
  while((dep = (dep_t *)pop(stk)) != NULL){

    // on sauvegarde l'état de l'échiquier
    save_end.kind = tab[dep->y][dep->x]->kind;
    save_end.color = tab[dep->y][dep->x]->color;
    // on sauvegarde uniquement les coordonnées de départ
    save_coord.x = tab_pieces[couleur][dep->kind].x;
    save_coord.y = tab_pieces[couleur][dep->kind].y;

    // on joue le coup
    move_chessman(tab[save_coord.y][save_coord.x], dep->x, dep->y);

    score = explore(prof + 1);

    // algorithme du minimax
    if(minimax(score, best_score, couleur)){
      best_score = score;
      if(prof == prof_start){
            best_dep.kind = dep->kind;
            best_dep.x = dep->x;
            best_dep.y = dep->y;
      }
    }

    // on annule le coup joué
    tab[dep->y][dep->x]->kind = save_end.kind;
    tab[dep->y][dep->x]->color = save_end.color;
    tab[save_coord.y][save_coord.x]->kind = dep->kind;
    tab[save_coord.y][save_coord.x]->color = couleur;
    tab_pieces[couleur][dep->kind].x = save_coord.x;
    tab_pieces[couleur][dep->kind].y = save_coord.y;
    if(save_end.kind != aucun){
      if(couleur == blanc){
          tab_pieces[noir][save_end.kind].x = dep->x;
          tab_pieces[noir][save_end.kind].y = dep->y;
      }else{
          tab_pieces[blanc][save_end.kind].x = dep->x;
          tab_pieces[blanc][save_end.kind].y = dep->y;
      }
    }

    // on efface l'élément dépilé de la mémoire
    free(dep);

  }
  return best_score;
}
```

Voici maintenant la fonction `list_all_mvt()` qui a été volontairement tronquée. Cette fonction attend en arguments la couleur du camp à traiter, ainsi qu'une pile. Comme nous l'avons vu dans l'article précédent, les coordonnées d'une pièce mangée sont à `(-1,-1)`, il ne faut donc pas traiter ces pièces.

```
void list_all_mvt(couleur_t couleur, stack_t stack)
{
  extern coord_t tab_pieces[noir+1][roi+1];
  extern piece_t *tab[8][8];

  dep_t dep;
```

```
  couleur_t couleur_adv;
  int x,y,i,j;

  if(couleur == blanc)
    couleur_adv = noir;
  else
    couleur_adv = blanc;

  // déplacements du roi
  if(tab_pieces[couleur][roi].x != -1 && tab_pieces[couleur][roi].y
!= -1){
    x = tab_pieces[couleur][roi].x;
    y = tab_pieces[couleur][roi].y;
    dep.kind = roi;
    for(j=0; j<=7; j++){
      x += move_roi_seq[j].x;
      y += move_roi_seq[j].y;
      if(is_coord_inside(x,y))
          // on ne va empiler que les coups possibles
          if((tab[y][x]->kind == aucun) || (tab[y][x]->color ==
couleur_adv)){
           dep.x = x;
           dep.y = y;
           // on empile le coup
           push(stack, &dep);
          }
    }
  }
```

La suite concerne les autres types de pièces. Penchez-vous sur le code des pions, qui est plus compliqué étant donné qu'un pion peut au départ avancer de deux cases, qu'il ne peut manger qu'en diagonale... Le principe reste toujours le même pour les autres types de pièces : le cavalier est analogue au roi (vu plus haut) ; la tour, le fou et la dame font partie de la même classe.

```
  // déplacements des pions
  for(i=pion1; i<=pion8 ; i++){
    if(tab_pieces[couleur][i].x == -1 && tab_pieces[couleur][i].y
== -1)
      continue;
    x=tab_pieces[couleur][i].x;
    y=tab_pieces[couleur][i].y;
    dep.kind = i;
    if(couleur == blanc){
      if(is_coord_inside(x,y-1))
          if((tab[y-1][x]->kind == aucun)){
            dep.x = x;
            dep.y = y-1;
            push(stack, &dep);
            if(y == 6){
              if(tab[y-2][x]->kind == aucun){
                dep.x = x;
                dep.y = y-2;
                push(stack, &dep);
              }
            }
          }
      if(is_coord_inside(x-1,y-1))
          if(tab[y-1][x-1]->color == couleur_adv){
            dep.x = x-1;
            dep.y = y-1;
            push(stack, &dep);
          }
      if(is_coord_inside(x+1,y-1))
          if(tab[y-1][x+1]->color == couleur_adv){
            dep.x = x+1;
            dep.y = y-1;
            push(stack, &dep);
          }
    }else{
      if(is_coord_inside(x,y+1))
          if((tab[y+1][x]->kind == aucun)){
            dep.x = x;
            dep.y = y+1;
            push(stack, &dep);
            if(y == 1)
              if(tab[y+2][x]->kind == aucun){
                dep.x = x;
                dep.y = y+2;
                push(stack, &dep);
              }
          }
      if(is_coord_inside(x-1,y+1))
          if(tab[y+1][x-1]->color == couleur_adv){
            dep.x = x-1;
            dep.y = y+1;
            push(stack, &dep);
          }
      if(is_coord_inside(x+1,y+1))
          if(tab[y+1][x+1]->color == couleur_adv){
            dep.x = x+1;
            dep.y = y+1;
            push(stack, &dep);
          }
    }
  }

  // déplacements des tours
  for(i=tour1; i<=tour2 ; i++){
    if(tab_pieces[couleur][i].x == -1 && tab_pieces[couleur][i].y
== -1)
      continue;
    x = tab_pieces[couleur][i].x;
    y = tab_pieces[couleur][i].y;
    dep.kind = i;
    for(j=0; j<=3; j++){
      x += move_tour[j].x;
      y += move_tour[j].y;
      while(is_coord_inside(x,y)){
          if(tab[y][x]->kind == aucun){
            dep.x = x;
            dep.y = y;
            push(stack, &dep);
            x += move_tour[j].x;
            y += move_tour[j].y;
          }else{
            break;
          }
      }
      if(is_coord_inside(x,y))
          if(tab[y][x]->color == couleur_adv){
            dep.x = x;
            dep.y = y;
            push(stack, &dep);
          }
      x = tab_pieces[couleur][i].x;
      y = tab_pieces[couleur][i].y;
    }
  }
        /* ici doit se trouver le code pour les autres pièces */
}
```

Limites (temps de calculs)

Vous l'aurez compris, le traitement récursif est très lourd si l'on doit parcourir tout l'arbre des situations possibles, même en limitant la profondeur d'exploration. Ce traitement n'est pas gênant pour un jeu comme le morpion, mais pour les échecs où l'on a en moyenne 35 coups possibles par tour et par couleur, on arrive très vite à des temps de calculs trop importants. Par exemple pour une profondeur 6, on a déjà des millions de situations à évaluer (35^6). C'est pourquoi il va falloir utiliser des techniques permettant de limiter cette exploration. L'*alpha/beta* est l'une d'entre elles, dont l'intérêt est de conserver une solution optimale. Diverses heuristiques associées permettent d'accroître encore l'efficacité de la recherche du coup à jouer, mais elles n'aboutissent pas forcément à la solution optimale. Nous discuterons de ceci dans la section sur les *Heuristiques Complémentaires* (p.29)

Heuristique d'évaluation de l'échiquier

Comme nous l'avons vu dans la section précédente, il est inconcevable de parcourir l'arbre des coups possibles d'une partie entière pour un jeu comme les échecs. Le rôle de la fonction d'évaluation de l'échiquier, ou plus généralement de l'état du jeu, est justement de limiter la profondeur d'exploration de l'arbre. En ce sens, elle est une des principales composantes de l'IA mise en place.

Pour implémenter l'heuristique dans le cas des échecs, nous allons définir des critères sur lesquels s'appuyer pour évaluer l'état d'un échiquier. Il nous faudra également grader ces critères en fonction de leur pertinence. Pour réaliser ceci, nous ferons intervenir la notion de score qui correspondra à une somme pondérée des points accumulés pour chaque critère. Cette pondération sera en rapport avec la pertinence des critères considérés.

Critères d'évaluation

Possession matérielle

Le premier des critères venant à l'esprit correspond bien sûr au nombre de pièces dont dispose chaque camp, avec en plus une prise en compte de l'importance de chaque type de pièce. Pour mettre en œuvre ce critère, nous associons simplement un nombre de points à chaque type de pièce, de manière à favoriser le camp dominant l'échiquier matériellement. On se référera à l'article précédent pour la déclaration de ces valeurs (sous forme de tableau).

Critère de sécurité des rois

Ce critère nous permet d'éviter que l'ordinateur se mette dans une situation d'échec. Il est en fait réalisé indirectement par le premier critère. En effet, il suffit d'associer au roi un grand nombre de points de manière à rendre les autres valeurs de pièces quasi négligeables face à elles. Il ne faut pas cependant pas qu'elle soit trop élevée (d'où le "quasi"), car le cas échéant, l'évaluation ne tiendrait en fait plus compte des autres pièces, allant ainsi à l'encontre de notre premier critère.

Possession du centre

La prise en compte de la position des pièces sur l'échiquier est sûrement le paramètre le plus dur à mettre au point. L'implémentation n'est pas trop délicate, mais la multiplicité des configurations possibles de l'échiquier rend l'étalonnage de ce critère très délicat.

L'objectif est de favoriser les positions centrales, influant sur le nombre de mouvements possibles. Considérons en effet l'exemple d'un fou : s'il se situe au bord de l'échiquier, sa liberté de mouvement est très amoindrie par rapport à une position centrale. Il perd ainsi deux "diagonales" (voire trois) de mouvements possibles. De plus, le critère influence la stratégie qui est *a priori* attaquante par la mise en valeur du centre.

Les bases étant fixées, il nous reste à discuter de l'importance accordée aux coordonnées verticales et horizontales (respectivement ordonnées et abscisses). Comme le déplacement se fait majoritairement vers l'avant, du fait que le camp adverse se situe en face, on construit notre évaluation par rapport à l'axe vertical, tout en faisant intervenir les abscisses à un degré moindre et seulement dans certaines situations. De cette façon, les lignes 4 et 5 seront favorisées (sauf pour le roi) et la position dans ces lignes ajoute éventuellement une bonification (cf l'implémentation en C plus bas – l'axe horizontal n'a pas ici été pris en compte).

L'univers des possibilités d'évaluation est ainsi agrandi par ce critère et non de manière négligeable. Vous pourrez, à votre souhait, modifier le nombre de points associé aux différentes positions des pièces sur l'échiquier (ainsi qu'aux autres critères), soit grâce à vos connaissances en ce jeu, soit par expérimentation lorsque vous jouerez face à l'ordinateur.

Nombre de mouvements possibles (degrés de liberté)

Ce dernier critère pris en considération dans l'élaboration du jeu d'échecs permet d'ajouter un peu de sophistication à notre fonction d'évaluation. On augmente ainsi la pertinence de cette heuristique, mais on évite aussi des situations de répétition qui peuvent se produire lorsque l'on joue consécutivement le même coup. En effet, l'ordinateur dans certaines situations peut ne plus évoluer et jouer toujours la même riposte tant que l'adversaire continue la répétition de son coup. Plus le nombre de paramètres (critères) est important, plus ces situations sont évitées.

Pour en revenir au critère portant sur la liberté de mouvement d'un camp, on peut le voir comme un vecteur de faveur envers le joueur orchestrant au mieux le combat qu'il livre à son adversaire.

Implémentation en C dans le jeu d'échecs

Voici la fonction d'évaluation retournant un entier. Elle s'appuie sur le tableau des coordonnées des pièces défini dans `OBJETS`.

```
int check_chessboard(void)
{
  extern coord_t tab_pieces[noir+1][roi+1];
  couleur_t score_noir = 0;
  couleur_t score_blanc = 0;
  int i;

  // score des noirs
  for(i=tour1; i<=tour2; i++){
    if(tab_pieces[noir][i].x == -1)
      continue;
    switch(tab_pieces[noir][i].y){
    case 6: case 7:
      score_noir += 50;
      break;
    case 0: case 1:
      score_noir += 10;
      break;
    case 2: case 5:
      score_noir += 120;
      break;
    case 3: case 4:
      score_noir += 150;
      break;
    }
  }

  for(i=cavalier1; i<=cavalier2; i++){
    if(tab_pieces[noir][i].x == -1)
      continue;
    switch(tab_pieces[noir][i].y){
    case 6: case 7:
      score_noir += 60;
      break;
    case 0: case 1:
      score_noir += 10;
      break;
    case 2: case 5:
      score_noir += 170;
      break;
    case 3: case 4:
      score_noir += 130;
      break;
    }
  }

  for(i=fou1; i<=fou2; i++){
    if(tab_pieces[noir][i].x == -1)
      continue;
    switch(tab_pieces[noir][i].y){
    case 6: case 7:
      score_noir += 60;
      break;
    case 1: case 0:
      score_noir += 10;
      break;
    case 2: case 5:
      score_noir += 150;
      break;
    case 3: case 4:
      score_noir += 200;
      break;
    }
  }

  if(tab_pieces[noir][dame].x != -1)
    switch(tab_pieces[noir][dame].y){
    case 6: case 7:
      score_noir += 80;
      break;
    case 0: case 1:
      score_noir += 100;
      break;
    case 2: case 5:
      score_noir += 200;
      break;
    case 3: case 4:
      score_noir += 300;
      break;
    }

  // valeur des pièces des noirs
  for(i=pion1; i<=roi; i++){
    if(tab_pieces[noir][i].x == -1)
      continue;
    score_noir += value_tab[i];
  }

  // nombre de positions possibles des noirs
  score_noir += 30*count_all_mvt(noir);


  // score des blancs

          /* ici doit se trouver le calcul du score des blancs
             qui est le symétrique de celui des noirs */

  return(score_blanc - score_noir);
}
```

La fonction suivante est analogue à `list_all_mvt()`, seulement, au lieu d'empiler des coups dans une pile, elle incrémente une variable. Il en résulte la satisfaction du critère sur les degrés de liberté d'un camp donné. On remarque néanmoins que le cas des pions n'est pas traité. Il n'a pas été pour autant tronqué pour cet article : nous avons volontairement fait le choix de ne pas prendre en compte leurs mouvements dans l'évaluation des degrés de liberté d'une couleur. En fait, leur rôle entre en compte lors du calcul du nombre de mouvements possibles des autres pièces, le limitant suivant leur position. De plus, il nous permet de nous affranchir d'un traitement dans la fonction d'évaluation qui est exécutée un très grand nombre de fois (une fois par feuille de l'arbre exploré). On peut donc voir ceci comme une optimisation du temps de traitement.

```
int count_all_mvt(couleur_t couleur)
{
  extern coord_t tab_pieces[noir+1][roi+1];
  extern piece_t *tab[8][8];

  couleur_t couleur_adv;
  int x,y,i,j;
  int cpt = 0;

  if(couleur == blanc)
    couleur_adv = noir;
  else
    couleur_adv = blanc;

  // déplacements du roi
  if(tab_pieces[couleur][roi].x != -1 && tab_pieces[couleur][roi].y != -1){
    x = tab_pieces[couleur][roi].x;
    y = tab_pieces[couleur][roi].y;
    for(j=0; j<=7; j++){
      x += move_roi_seq[j].x;
      y += move_roi_seq[j].y;
      if(is_coord_inside(x,y))
          if((tab[y][x]->kind == aucun) || (tab[y][x]->color ==
couleur_adv))
            cpt++;
    }
  }

  // déplacements des tours
  for(i=tour1; i<=tour2 ; i++){
    if(tab_pieces[couleur][i].x == -1 && tab_pieces[couleur][i].y == -1)
      continue;
    x = tab_pieces[couleur][i].x;
    y = tab_pieces[couleur][i].y;
    for(j=0; j<=3; j++){
      x += move_tour[j].x;
      y += move_tour[j].y;
      while(is_coord_inside(x,y)){
          if(tab[y][x]->kind == aucun){
            cpt++;
            x += move_tour[j].x;
```

```
            y += move_tour[j].y;
          }else{
            break;
          }
      }
      if(is_coord_inside(x,y))
          if(tab[y][x]->color == couleur_adv)
            cpt++;
      x = tab_pieces[couleur][i].x;
      y = tab_pieces[couleur][i].y;
    }
  }

  // déplacements des cavaliers
  for(i=cavalier1; i<=cavalier2 ; i++){
    if(tab_pieces[couleur][i].x == -1 && tab_pieces[couleur][i].y == -1)
      continue;
    x = tab_pieces[couleur][i].x;
    y = tab_pieces[couleur][i].y;
    for(j=0; j<=7; j++){
      x += move_cav_seq[j].x;
      y += move_cav_seq[j].y;
      if(is_coord_inside(x,y))
          if((tab[y][x]->kind == aucun) || (tab[y][x]->color ==
couleur_adv))
            cpt++;
    }
  }

  // déplacements des fou
  for(i=fou1; i<=fou2 ; i++){
    if(tab_pieces[couleur][i].x == -1 && tab_pieces[couleur][i].y == -1)
      continue;
    x = tab_pieces[couleur][i].x;
    y = tab_pieces[couleur][i].y;
    for(j=0; j<=3; j++){
      x += move_fou[j].x;
      y += move_fou[j].y;
      while(is_coord_inside(x,y)){
          if(tab[y][x]->kind == aucun){
            cpt++;
            x += move_fou[j].x;
            y += move_fou[j].y;
          }else{
            break;
          }
      }
      if(is_coord_inside(x,y))
          if(tab[y][x]->color == couleur_adv)
            cpt++;
      x = tab_pieces[couleur][i].x;
      y = tab_pieces[couleur][i].y;
    }
  }

  // déplacement de la dame
  if(tab_pieces[couleur][dame].x != -1 && tab_pieces[couleur][dame].y != -
1){
    x = tab_pieces[couleur][dame].x;
    y = tab_pieces[couleur][dame].y;
    for(j=0; j<=7; j++){
      x += move_d_r[j].x;
      y += move_d_r[j].y;
      while(is_coord_inside(x,y)){
          if(tab[y][x]->kind == aucun){
            cpt++;
            x += move_d_r[j].x;
            y += move_d_r[j].y;
          }else{
            break;
          }
      }
      if(is_coord_inside(x,y))
          if(tab[y][x]->color == couleur_adv)
            cpt++;
      x = tab_pieces[couleur][dame].x;
      y = tab_pieces[couleur][dame].y;
    }
  }

  return cpt;
}
```

Les heuristiques en général

L'écriture d'une heuristique d'évaluation est un exercice assez difficile, qui conditionne les performances du programme. En effet, plus l'heuristique est précise, c'est-à-dire plus elle prédit correctement l'issue de la partie, moins la recherche dans l'arbre doit être profonde. Si on était capable de prédire le gagnant d'une partie d'échecs dès le quatrième coup, il serait alors inutile d'étudier un arbre de 8 coups de profondeur.

Dans un jeu comme Othello, on peut procéder d'une façon assez similaire à celle proposée au-dessus pour les échecs. La possession matérielle est en effet un critère important, en particulier en fin de partie (car le but d'Othello est d'avoir plus de pions de notre couleur que l'adversaire). Contrairement aux échecs, il est plutôt conseillé de ne pas avoir trop de pions en début de partie. La position des pions est aussi assez importante car les coins ne sont pas prenables, par exemple (on a donc une sorte d'équivalent pour Othello du contrôle du centre aux échecs). De même, les degrés de liberté sont assez importants. La notion de frontière est aussi très cruciale, et assez spécifique à Othello : il s'agit en effet de minimiser les pions situés en bordure, non pas du plateau de jeu, mais de la zone centrale occupée en premier. Ces pions sont en effet très vulnérables.

Il est assez difficile d'aller plus loin "à la main", c'est pourquoi des chercheurs ont proposé depuis quelques années d'utiliser toute la panoplie de l'apprentissage automatique pour faire construire par l'ordinateur des heuristiques d'évaluation [2]. L'un des pionniers dans ce domaine est Gerald Tesauro, qui a réalisé le programme TD-Gammon [3]. Celui-ci joue extrêmement bien au backgammon grâce à une heuristique calculée par un réseau de neurones. L'heuristique est construite totalement automatiquement par le programme. Le plus impressionnant peut-être est que l'apprentissage se fait seul : TD-Gammon apprend en jouant contre lui-même !

D'autres chercheurs avaient initié avant Tesauro des techniques similaires, mais en choisissant des outils plus simples (les réseaux de neurones calculent des fonctions non linéaires assez complexes), aidés par une représentation évoluée de la situation du jeu. En effet, dans TD-Gammon, le réseau de neurones reçoit une simple description de l'état du jeu, sans pré-traitement. Dans les approches antérieures, l'idée était plutôt de découvrir dans l'état du jeu des configurations intéressantes, qui donnent un avantage à l'un ou l'autre des joueurs, voire plus simplement de régler les pondérations entre les différents critères évolués tels que la possession matérielle, l'occupation du centre, etc. Pour ce faire, l'idée de base consiste à utiliser une grande base de données de parties de grands maîtres sur lesquelles

on calcule des statistiques évoluées. Un des plus grands succès dans ce domaine vient d'une combinaison des idées de Tesauro (l'apprentissage seul) avec les idées antérieures (la recherche de configurations intéressantes et l'apprentissage par l'exemple). Il s'agit du programme Logistello de Michael Buro [4] (pour le jeu Othello). L'idée fondatrice est de choisir un très grand nombre de configurations élémentaires (par exemple, la position d'un pion blanc dans une case donnée), de façon quasi mécanique, puis de faire déterminer l'importance de chaque configuration automatiquement par l'affrontement du programme avec lui-même et l'étude de parties de grands maîtres. En 1997, Logistello a battu 6 fois de suite Takeshi Murakami, le champion du monde de l'époque...

Malheureusement, l'apprentissage automatique n'a pas résolu tous les problèmes. En fait, il a fait ses preuves pour Othello, pour le backgammon, ou encore pour les dames anglaises (une variante des dames sur un plateau 8x8, avec 12 pièces par joueur). Pour ce dernier exemple, il est intéressant de noter que le meilleur programme, Chinook [5], qui est champion du monde, possède une heuristique réglée à la main, après quelques tentatives infructueuses de réglage automatique. Les heuristiques apprises automatiquement ont obtenu un excellent niveau, mais pas suffisant pour battre Chinook. Il s'agit donc en quelque sorte du cas frontière, car pour les échecs ou pour le Go, l'apprentissage automatique ne semble pas encore capable de rivaliser avec les experts humains.

Comment diminuer le coût en temps de calcul

Étant donné une fonction d'évaluation, l'algorithme du Minimax calcule la solution optimale et donne donc la meilleure suite de coups possibles. Cependant, le temps de calcul est souvent trop grand. Pour rendre le traitement plus efficace, on utilise diverses variantes du Minimax. Nous commencerons par présenter alpha/beta, une variante qui peut accélérer radicalement le minimax tout en conservant une solution optimale. Nous enchaînerons sur une heuristique permettant justement à alpha/beta d'exprimer toute sa puissance en fixant l'ordre d'exploration des coups. Nous terminerons par des heuristiques plus délicates d'emploi, car entraînant parfois le calcul d'une solution non optimale.

L'élagage alpha/beta

Explication théorique

Cet algorithme permet de supprimer des branches dans le parcours de l'arbre, par une considération logique toujours sur deux niveaux. On utilise deux bornes distinctes : alpha, pour supprimer des branches lors de l'exploration des coups des noirs et beta dans le cas des blancs. Le fait d'agir sur deux niveaux consécutifs de l'arbre nous permet de conserver des informations des sous-arbres du niveau inférieur et de les utiliser lors de l'exploration des sous-arbres frères via le niveau supérieur. On ajoute en quelque sorte de l'intelligence au minimax.

Prenons le cas de la borne *alpha* (cf **Figure 9**), celui de *beta* s'en déduit par simple analogie. Le but est de maintenir à jour une variable `alpha` (dans un nœud blanc), qui prend la valeur la plus élevée des fils (nœuds noirs) qui ont déjà été visités à ce niveau. Cette valeur correspond au coup le plus favorable pour l'IA pour l'instant. En effet, à un niveau des blancs dans l'arbre, la convention choisie est de garder le coup qui correspond au score le plus positif des fils, c'est-à-dire le coup des blancs qui conduit à l'échiquier le plus favorable pour le joueur concerné. Pour explorer un nouveau fils noir, on doit étudier les nœuds blancs qui sont accessibles depuis le nœud noir concerné. Dès lors que l'on trouve pendant cette exploration, qui concerne donc les coups possibles pour les noirs, une valeur inférieure à la borne *alpha*, il ne sert à rien de visiter la suite du sous-arbre en question. En effet, toute valeur trouvée serait : soit supérieure, auquel cas elle ne serait pas sélectionnée (aux niveaux des noirs on choisit la plus petite valeur) ; soit inférieure, auquel cas elle serait *a fortiori* inférieure à alpha et ne constituerait donc pas un nœud sélectionné au niveau supérieur, celui des blancs. Ce mécanisme est appelé "coupe *alpha*". La "coupe *beta*" correspond au même mécanisme pour les noirs. Dans ce cas, on teste pour effectuer la coupe si les scores associés aux coups blancs sont supérieurs à la borne *beta*.

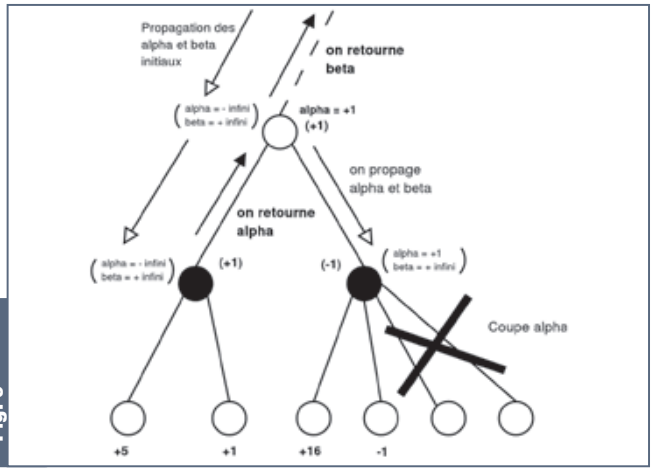

Fig. 9

Comme nous pouvons le voir sur la **figure 9**, au dernier niveau dans le sous-arbre de gauche, on choisit la valeur +1 (score le plus petit), ensuite on explore le sous-arbre frère, et on tombe sur la valeur –1, qui est inférieure à +1 ; par conséquent il devient inutile d'explorer la suite.

Terminons cette présentation sur quelques remarques importantes. Tout d'abord, quel choix aurait-on dû faire si on avait eu le score +1 à la place du –1 ? La réponse est simple, si on continuait le parcours, on ne pourrait au mieux que conserver ce score qui serait le même que dans le nœud frère : on aurait donc deux situations équivalentes aux yeux de notre fonction d'évaluation. Il convient donc d'appliquer une coupe *alpha* dans le cas de l'égalité.

D'autre part, on remarque que les coupes peuvent très bien se passer en début d'exploration à un niveau donné, comme

à la fin, auquel cas le gain devient nul. C'est pour cela que nous allons faire intervenir des heuristiques favorisant la première situation.

Il faut aussi noter qu'une borne alpha (ou beta) peut être utilisée pour faire une coupe deux niveaux plus bas, comme dans la présentation de l'algorithme mais aussi quatre, six, etc. niveaux plus profond dans l'arbre.

D'autre part, les coupes ne s'effectueront jamais au premier niveau (i.e. à partir de la racine). En effet, le seul moyen de faire une coupe est d'avoir un nœud parent qui pilote la recherche en donnant une valeur à alpha ou à beta. La racine n'a pas de nœud parent (par définition) et ne peut donc pas déclencher de coupes pertinentes.

Illustration sur le morpion

Considérons l'arbre du morpion représenté à la **figure 7** et reproduit dans la **figure 10**. Alpha est initialisé à -infini et beta à +infini avant toute exploration. On commence par explorer la branche la plus à gauche, jusqu'à la feuille correspondant à la victoire du joueur A.

On doit remonter deux fois pour atteindre un nœud avec une alternative (le nœud B). Cependant, celle-ci n'est pas concernée par alpha/beta. En effet, le nœud marqué A sur la **figure 10** n'a qu'un seul fils et on est donc obligé de l'explorer avant même de pouvoir faire une éventuelle coupe. Par contre, quand on termine l'exploration du nœud marqué B, on obtient une valeur de 0 pour alpha. Or, l'exploration de la première branche partant du nœud marqué C donne un score de 0 (pour le nœud D). On peut donc faire une coupe alpha car si l'exploration de la deuxième branche partant de C donne un score inférieur à 0, le nœud C aura donc une valeur inférieure à 0 et l'étape max au-dessus sélectionnera le nœud B. Par contre, si le score est supérieur à 0, l'étape mini au nœud C sélectionnera la première branche.

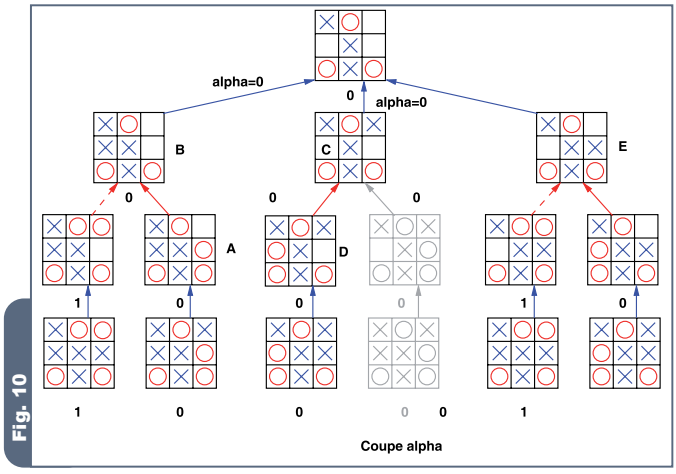

Fig. 10

Cette situation favorable ne se reproduit pas pour les branches partant du nœud E. En effet, pour ce nœud, nous explorons d'abord une branche qui donne un score de 1, supérieur à alpha et nous devons donc explorer la branche suivante. On voit donc bien ici l'importance de l'ordre d'évaluation des coups sur l'efficacité d'alpha/beta.

Implémentation en C dans le jeu d'échecs

Avant de voir le code, notons quelques points importants. Tout d'abord, le processus de coupe est à mettre en œuvre à chaque niveau de l'arbre (sauf au premier), il faut donc passer en argument de la fonction récursive d'exploration les variables alpha/beta de manière à propager les bonnes valeurs. De plus, on doit initialiser respectivement *alpha* et *beta* à *-infini* et *+infini*, de manière à ce que la mise à jour de *alpha* ou de *beta* se fasse bien au tout début avec une valeur réellement issue d'une évaluation. En effet, la valeur -infini ne peut pas déclencher de coupe alpha car l'évaluation sera toujours plus grande et symétriquement la valeur +infini ne peut pas déclencher de coupe beta car l'évaluation sera toujours plus petite. Par contre, dès qu'un appel récursif a remonté (cf la suite) une valeur pour alpha ou beta, tous les appels suivants depuis le nœud courant peuvent utiliser la nouvelle valeur pour faire des coupes.

Il faut prendre garde de mettre à jour, lors de la remontée "des" bornes, seulement celles correspondant à la couleur du niveau (*alpha* pour les blancs), de manière à éviter de propager une borne (*beta* dans l'exemple de la Figure 9) invalide dans le sous-arbre frère. Il faut propager une borne infinie pour qu'il n'y ait pas de conflit dans les processus de coupes, sinon on aboutirait forcément à un élagage excessif et hasardeux (le résultat serait donc totalement erroné). On voit ainsi qu'il est inutile de faire remonter les deux bornes. Dans un nœud noir, on renverra *alpha* et dans un nœud blanc on renverra *beta* uniquement. En fait, c'est l'évaluation du nœud qui est remontée et interprétée en fonction du contexte.

La fonction `explore()` change donc de définition et de prototype. Les ajouts dans le code propre à l'alpha/beta ont été signalés. Voici d'abord la définition de la structure de retour qui se trouve dans `ia.h`.

```
/* ia.h */

#define ALPHA_INIT -100000  // correspond à - infini
#define BETA_INIT 100000    // correspond à + infini

typedef struct _ret_t{
  int score;
  int alpha_or_beta;
}ret_t;

ret_t *explore(int prof, int _alpha, int _beta);

/* fonction récursive de ia.c */

ret_t *explore(int prof, int _alpha, int _beta)
{
  extern piece_t *tab[8][8];
  extern coord_t tab_pieces[noir+1][roi+1];
  extern dep_t best_dep;

  int alpha = _alpha;
  int beta = _beta;

  stack_t stk;
  dep_t *dep;
  int best_score;
  ret_t *ret;
  ret_t *to_send;
  couleur_t couleur;
  piece_t save_end;
  coord_t save_coord;
```

```
// choix de la couleur en fonction de la profondeur
if(prof%2 == 0){
  couleur = blanc;
  best_score = -100000;
}else{
  couleur = noir;
  best_score = 100000;
}

// fin de l'exploration
if(prof == prof_max){
  to_send = malloc(sizeof(ret_t));
  to_send->score = check_chessboard();
  to_send->alpha_or_beta = 0; // ce champ n'a ici pas d'importance
  return(to_send);
}

// à un niveau donné on liste tous les coups possibles de la couleur concernée
stk = create_stack(copy_dep);
list_all_mvt(couleur,stk);

// tant que la pile des coups n'est pas vide on joue le coup dépilé
while((dep = (dep_t *)pop(stk)) != NULL){

  // on sauvegarde l'état de l'échiquier
  save_end.kind = tab[dep->y][dep->x]->kind;
  save_end.color = tab[dep->y][dep->x]->color;
  // on sauvegarde uniquement les coordonnées de départ
  save_coord.x = tab_pieces[couleur][dep->kind].x;
  save_coord.y = tab_pieces[couleur][dep->kind].y;

  // on joue le coup
  move_chessman(tab[save_coord.y][save_coord.x], dep->x, dep->y);

  ret = explore(prof + 1, alpha, beta);

  // fait partie de l'implémentation de l'alpha/beta
  if((prof != prof_start) && (prof != prof_max - 1)){
    if(couleur == noir)
          beta = ret->alpha_or_beta;
    else
          alpha = ret->alpha_or_beta;
  }

  // algorithme du minimax
  if(minimax(ret->score, best_score, couleur)){
    best_score = ret->score;
    if(prof == prof_start){
          best_dep.kind = dep->kind;
          best_dep.x = dep->x;
          best_dep.y = dep->y;
    }
  }

  // on libère la structure ret dont on n'a plus besoin
  free(ret);

  // on annule le coup joué
  tab[dep->y][dep->x]->kind = save_end.kind;
  tab[dep->y][dep->x]->color = save_end.color;
  tab[save_coord.y][save_coord.x]->kind = dep->kind;
  tab[save_coord.y][save_coord.x]->color = couleur;
  tab_pieces[couleur][dep->kind].x = save_coord.x;
  tab_pieces[couleur][dep->kind].y = save_coord.y;
  if(save_end.kind != aucun){
    if(couleur == blanc){
          tab_pieces[noir][save_end.kind].x = dep->x;
          tab_pieces[noir][save_end.kind].y = dep->y;
    }else{
          tab_pieces[blanc][save_end.kind].x = dep->x;
          tab_pieces[blanc][save_end.kind].y = dep->y;
    }
  }

  // on efface l'élément dépilé de la mémoire
  free(dep);

  // implémentation de la coupe alpha ou beta
  if((couleur == noir) && (best_score <= alpha) && (prof != prof_start))
      break;
    if((couleur == blanc) && (best_score >= beta) && (prof != prof_start))
      break;
  }

  /* si une coupe a été réalisée la pile n'est pas vide
     on doit donc la vider car la saturation mémoire se fait très
     vite ressentir */
  empty(stk);

  // dernière partie de l'implémentation de l'alpha/beta
  to_send = malloc(sizeof(ret_t));
  to_send->score = best_score;

  if(couleur == noir){
    if(alpha < best_score)
      to_send->alpha_or_beta = best_score;
    else
      to_send->alpha_or_beta = alpha;
  }else{
    if(beta > best_score)
      to_send->alpha_or_beta = best_score;
    else
      to_send->alpha_or_beta = beta;
  }

  return to_send;
}
```

Et voici maintenant la fonction restante de `ARBITRE` qui initie la récursivité en fonction de la couleur de l'IA.

```
/* extrait de ia.c */

void ia_play(int prof)
{
  // exportées de ia.c
  extern int prof_max;
  extern int prof_start;

  ret_t *tmp;

  if(ia_color == blanc){
    prof_start = 0;
    prof_max = prof;
  }else{
    prof_start = 1;
    prof_max = prof + 1;
  }

  tmp = explore(prof_start, ALPHA_INIT, BETA_INIT);
  free(tmp);
}
```

Nous verrons dans le prochain article, en finissant la présentation du package `IHM`, comment l'interface graphique appelle cette fonction. En attendant, vous trouverez des éléments de réponse dans l'article précédent.

Heuristiques complémentaires

Ordonner la recherche des coups

Il y a peu de choses à dire concernant cette heuristique dans un premier temps (nous reviendrons sur ce point dans la section V.2.). Elle ne demande aucune implémentation supplémentaire mais est cependant cruciale si l'on souhaite un alpha/beta performant. En effet, il suffit de classer les branches par ordre d'importance. Il faudra donc traiter en premier les pièces les plus aptes à nous mener à des situations intéressantes. Par exemple, les déplacements de la dame sont la plupart du temps très importants, on pourra donc commencer l'exploration des coups avec ceux de la dame, ceci de manière à favoriser

les coupes. Référez-vous à la section sur l'implémentation pour l'ordre d'exploration choisi et à la section sur *l'approfondissement itéractif* (plus loin). pour une alternative plus complexe.

Le Killer Heuristic

Le concept de cette heuristique est de se souvenir, lors de l'exploration de l'arbre, d'un coup très fort, que l'on tentera de reproduire dans le sous-arbre frère. Dans beaucoup de cas, si ce coup est possible, on aboutira aussi à une situation très intéressante, ce qui favorisera donc encore le processus de coupe. On pourra bien sûr conserver plusieurs coups *killers* de manière à accentuer l'effet de l'heuristique. Cependant, il est à noter que le gain de performances induit par le *killer heuristic* est très inférieur à celui qu'amène l'alpha/beta correctement "ordonné" dans le minimax de base (cette heuristique n'a pas été implémentée dans le programme).

D'autres heuristiques

Pour conclure sur cette section, on peut aussi trouver des heuristiques "destructrices" (i.e. n'aboutissant pas forcément à un résultat optimal). Il existe par exemple des techniques statistiques permettant de réaliser des coupes proches similaires à celles de l'alpha/beta, mais plus brutales, au sens où on a une probabilité non nulle de se tromper, c'est-à-dire de couper une branche menant au résultat optimal. Les programmes Logistello et Chinook utilisent de telles techniques. On trouvera une description théorique d'une telle technique dans l'article ProbCut de Michael Buro [4].

Pour aller plus loin...

Diverses techniques permettent d'améliorer l'algorithme d'exploration et donc de descendre plus loin dans l'arbre, ce qui a pour conséquence finale de rendre le programme plus intelligent. Ces techniques sont assez complexes et n'ont pas été mises en œuvre dans le programme décrit dans la série d'articles sur le jeu d'échecs.

Les tables de transpositions

Une idée très simple mais fructueuse consiste à utiliser autant la mémoire de l'ordinateur que sa puissance de calcul. En effet, dans la plupart des jeux de stratégie, certaines suites de coups distinctes conduisent à des états identiques. On pense bien entendu en premier lieu à un simple changement dans l'ordre des coups, mais d'autres situations un peu plus complexes peuvent survenir.

Il suffit de regarder la **figure 7** pour constater que les six feuilles de l'arbre correspondent en fait à seulement 3 configurations du jeu. L'idée est donc d'essayer de ne pas dupliquer les efforts.

Si le programme évalue un état donné du jeu par un long parcours de l'arbre des coups possibles à partir de cet état, on peut gagner énormément de temps en s'apercevant qu'une nouvelle suite de coups vient de nous conduire dans exactement le même état.

On peut d'ailleurs gagner ponctuellement en profondeur de façon quasi gratuite, car rien ne dit que la nouvelle suite de coups a la même longueur que l'ancienne.

Une table de transpositions est une sorte de base de données qui est utilisée pour stocker les résultats de l'évaluation d'une configuration du jeu. En général, elle est implémentée grâce à une table de hachage car le temps d'accès, d'insertion et de suppression est *grosso modo* constant, à condition d'avoir une bonne fonction de hachage. Dans notre cas, la fonction doit transformer une configuration du jeu en un entier, ce qui demande un travail spécifique pour chaque jeu.

A chaque fois qu'on est amené à évaluer une nouvelle configuration du jeu, on commence par tester si la table de transposition la contient. Dans ce cas, aucun nouveau calcul n'est nécessaire. Dans le cas contraire, on peut sauvegarder le résultat de l'évaluation dans la table. Pour éviter que celle-ci ne grossisse trop, un mécanisme de vidange automatique s'implémente facilement, par exemple en supprimant les configurations qui n'ont pas été utilisées depuis longtemps.

Les tables de transpositions servent aussi à stocker les ouvertures et les fins de partie. En effet, le début de partie est en général très difficile pour l'ordinateur car il faudrait en théorie chercher très loin dans l'arbre du jeu pour pouvoir choisir des coups optimaux. Les joueurs humains sont ici aidés par une grande expérience du jeu qui leur permet de mémoriser des débuts standards, appelés ouvertures aux échecs. Pour aider l'ordinateur, on stocke dans une table de transposition des ouvertures classiques.

En fin de partie, l'ordinateur est assez puissant pour résoudre de façon formelle le jeu, c'est-à-dire pour se passer de l'heuristique d'évaluation et descendre jusqu'aux feuilles de l'arbre. L'idée des bibliothèques de fin de partie est de pré-calculer sur des gros calculateurs le maximum de fin de parties pour reculer la limite de résolution exacte du problème. On peut par exemple utiliser 2 mois de calculs sur un gros calculateur parallèle pour évaluer un certain nombre de fins de partie classiques puis stocker les évaluations obtenues dans une table de transposition qui sera elle utilisable sur un ordinateur "normal".

L'approfondissement itératif

Comme nous l'avons vu plus haut, les performances de alpha/beta dépendent fortement de l'ordre dans lequel les coups sont étudiés. Une technique particulièrement fructueuse pour choisir un ordre consiste à pratiquer l'approfondissement itératif (*iterative deepening*) de l'exploration de l'arbre.

L'idée est en fait d'utiliser les résultats de l'exploration à la profondeur k-2 pour guider la recherche à la profondeur k. Plus précisément, on commence par parcourir l'arbre du jeu à la profondeur k-2, en appliquant toutes les techniques déjà présentées, en particulier alpha/beta, avec

un ordre heuristique classique pour l'étude des coups. On obtient de cette manière une valeur pour chaque configuration du jeu explorée, stockée dans une table de transposition. Ensuite, on relance une exploration, mais avec une profondeur k. On repasse bien évidemment par les mêmes configurations, mais on doit recalculer les évaluations car un score obtenu pour une profondeur donnée ne renseigne pas en général sur le score qu'on obtiendra avec une autre profondeur.

Par contre, et toute l'astuce est là, les scores obtenus lors de l'exploration précédente permettent d'ordonner les coups. Par exemple, à un niveau blanc, les différentes configurations filles, chacune associée à un coup, ont des scores calculés avec l'exploration k-2. Si on jouait avec cette évaluation, on choisirait la meilleure configuration fille. On va donc commencer par explorer cette configuration. On procède de même à un niveau noir (sauf qu'on choisit la configuration fille avec le plus mauvais score). En général, la première blanche explorée remonte une excellente valeur pour alpha ou beta (selon le niveau étudié) qui déclenche très rapidement une coupe.

Il est légitime cependant de s'interroger sur le gain réel apporté par cette technique. En effet, elle consiste tout de même à refaire un grand nombre de calcul. De plus, il est fréquent de l'utiliser itérativement, c'est-à-dire en commençant à k=2, puis en passant à k=4, k=6, etc. En fait, la combinatoire nous sauve. En effet, si on compte b coups en moyenne par niveau, l'exploration d'un arbre de profondeur k correspond à l'étude de $1+b+b^2+\ldots+b^k$ configurations, c'est-à-dire $\frac{(b^{k+1}-1)}{(b-1)}$ nœuds. Le passage à k+2 correspond donc à l'ajout à cette somme de la valeur $b^{k+1}+b^{k+2}$. Or, comme b est en général assez grand (par exemple 35 pour les échecs), $\frac{(b^{k+1}-1)}{(b-1)}<b^{k+1}$ et même $\frac{(b^{k+1}-1)}{(b-1)}\approx b^k$. De plus, b^{k+2} est très grand devant b^{k+1} (b fois plus grand) et encore plus devant b^k. De ce fait, le calcul des évaluations avec une profondeur k+2 est dominé par $b^{k+1}+b^{k+2}$. Le temps de calcul correspondant à l'exploration à la profondeur k est donc négligeable devant celui consacré à la profondeur k+2. Pour fixer les idées, 35 à la puissance 8 vaut plus de 2 251 milliards, alors que $\frac{(35^9-1)}{(35-1)}$ vaut à peine 2 milliards, en fait près de 1200 fois moins. De plus, l'impact sur alpha/beta est tel qu'on observe en général un temps de calcul total bien plus faible avec l'approfondissement itératif qu'avec une exploration directe à la profondeur maximale !

Conclusion

Malgré sa longueur et sa relative complexité, cet article n'a fait qu'aborder le domaine passionnant des jeux de stratégie. De nombreuses améliorations d'alpha/beta (NegaScout, MTD(f), etc.) sont par exemple envisageables. L'étude des heuristiques d'évaluation, des coupes statistiques et des dizaines d'astuces conduisant à la construction d'un programme comme Deep Blue [6], capable de battre le champion du monde en titre, pourrait nous occuper des années.

De façon plus terre à terre, nous renvoyons par exemple à [7] pour la présentation de certaines de ces techniques. Le lecteur courageux maîtrisant le code présenté dans le présent article pourra tenter de l'améliorer, en s'inspirant par exemple du vénérable GNU Chess [8].

Notons pour conclure que le domaine de l'IA des jeux de stratégie fait toujours l'objet d'une recherche très active et que les chercheurs les plus ambitieux comme Michael Buro songent maintenant à s'attaquer aux jeux de stratégie en temps réel.

Concernant l'implémentation du jeu d'échecs proposée dans la série d'articles dans laquelle nous nous inscrivons, une dernière étape est encore nécessaire.

Nous terminerons donc dans le prochain article la conception du jeu, avec entre autres les fonctions détectant les positions d'échec, d'échec et mat, ainsi que la fin de l'implémentation de l'interface graphique avec GTK+.

Eric Lacombe
tuxico@ifrance.com
Elève ingénieur à l'INSA de Toulouse

Fabrice Rossi
Fabrice.Rossi@apiacoa.org
Equipe Axis
INRIA

Références

[1] Site de freeciv : http://www.freeciv.org/

[2] Jay Scott maintient des pages Web très intéressantes concernant les applications de l'apprentissage automatique aux jeux de stratégie, à l'URL http://satirist.org/learn-game/.

[3] Une web-bibliographie sur TD-Gammon de Gerald Tesauro est disponible à l'URL : http://satirist.org/learn-game/systems/gammon/td-gammon.html

[4] La page Web de Michael Buro (http://www.cs.ualberta.ca/~mburo/) contient des liens vers la plupart de ses articles sur l'apprentissage automatique appliqué aux jeux de stratégie en général, et à Othello en particulier (Logistello).

[5] Site Web de Chinook : http://www.cs.ualberta.ca/~chinook/

[6] Site de Deep Blue chez IBM Research : http://www.research.ibm.com/deepblue/

[7] Une série d'articles sur l'IA des échecs est disponible sur Gamedev à l'URL http://www.gamedev.net/reference/list.asp?categoryid=18#63 (Articles Chess Programming de I à VI)

[8] Site de GNU Chess : http://www.gnu.org/software/chess/chess.html

Creative Commons

Paternité - Pas d'Utilisation Commerciale - Pas de Modification 2.0

Creative Commons n'est pas un cabinet d'avocats et ne fournit pas de services de conseil juridique. La distribution de la présente version de ce contrat ne crée aucune relation juridique entre les parties au contrat présenté ci-après et Creative Commons. Creative Commons fournit cette offre de contrat-type en l'état, à seule fin d'information. Creative Commons ne saurait être tenu responsable des éventuels préjudices résultant du contenu ou de l'utilisation de ce contrat.

Contrat

L'Oeuvre (telle que définie ci-dessous) est mise à disposition selon les termes du présent contrat appelé Contrat Public Creative Commons (dénommé ici « CPCC » ou « Contrat »). L'Oeuvre est protégée par le droit de la propriété littéraire et artistique (droit d'auteur, droits voisins, droits des producteurs de bases de données) ou toute autre loi applicable. Toute utilisation de l'Oeuvre autrement qu'explicitement autorisée selon ce Contrat ou le droit applicable est interdite.

L'exercice sur l'Oeuvre de tout droit proposé par le présent contrat vaut acceptation de celui-ci. Selon les termes et les obligations du présent contrat, la partie Offrante propose à la partie Acceptante l'exercice de certains droits présentés ci-après, et l'Acceptant en approuve les termes et conditions d'utilisation.

1. Définitions

a. **« Oeuvre »** : oeuvre de l'esprit protégeable par le droit de la propriété littéraire et artistique ou toute loi applicable et qui est mise à disposition selon les termes du présent Contrat.
b. **« Oeuvre dite Collective »** : une oeuvre dans laquelle l'oeuvre, dans sa forme intégrale et non modifiée, est assemblée en un ensemble collectif avec d'autres contributions qui constituent en elles-mêmes des oeuvres séparées et indépendantes. Constituent notamment des Oeuvres dites Collectives les publications périodiques, les anthologies ou les encyclopédies. Aux termes de la présente autorisation, une oeuvre qui constitue une Oeuvre dite Collective ne sera pas considérée comme une Oeuvre dite Dérivée (telle que définie ci-après).
c. **« Oeuvre dite Dérivée »** : une oeuvre créée soit à partir de l'Oeuvre seule, soit à partir de l'Oeuvre et d'autres oeuvres préexistantes. Constituent notamment des Oeuvres dites Dérivées les traductions, les arrangements musicaux, les adaptations théâtrales, littéraires ou cinématographiques, les enregistrements sonores, les reproductions par un art ou un procédé quelconque, les résumés, ou toute autre forme sous laquelle l'Oeuvre puisse être remaniée, modifiée, transformée ou adaptée, à l'exception d'une oeuvre qui constitue une Oeuvre dite Collective. Une Oeuvre dite Collective ne sera pas considérée comme une Oeuvre dite Dérivée aux termes du présent Contrat. Dans le cas où l'Oeuvre serait une composition musicale ou un enregistrement sonore, la synchronisation de l'oeuvre avec une image animée sera considérée comme une Oeuvre dite Dérivée pour les propos de ce Contrat.
d. **« Auteur original »** : la ou les personnes physiques qui ont créé l'Oeuvre.
e. **« Offrant »** : la ou les personne(s) physique(s) ou morale(s) qui proposent la mise à disposition de l'Oeuvre selon les termes du présent Contrat.
f. **« Acceptant »** : la personne physique ou morale qui accepte le présent contrat et exerce des droits sans en avoir violé les termes au préalable ou qui a reçu l'autorisation expresse de l'Offrant d'exercer des droits dans le cadre du présent contrat malgré une précédente violation de ce contrat.

2. Exceptions aux droits exclusifs. Aucune disposition de ce contrat n'a pour intention de réduire, limiter ou restreindre les prérogatives issues des exceptions aux droits, de l'épuisement des droits ou d'autres limitations aux droits exclusifs des ayants droit selon le droit de la propriété littéraire et artistique ou les autres lois applicables.

3. Autorisation. Soumis aux termes et conditions définis dans cette autorisation, et ceci pendant toute la durée de protection de l'Oeuvre par le droit de la propriété littéraire et artistique ou le droit applicable, l'Offrant accorde à l'Acceptant l'autorisation mondiale d'exercer à titre gratuit et non exclusif les droits suivants :

a. reproduire l'Oeuvre, incorporer l'Oeuvre dans une ou plusieurs Oeuvres dites Collectives et reproduire l'Oeuvre telle qu'incorporée dans lesdites Oeuvres dites Collectives;
b. distribuer des exemplaires ou enregistrements, présenter, représenter ou communiquer l'Oeuvre au public par tout procédé technique, y compris incorporée dans des Oeuvres Collectives;
c. lorsque l'Oeuvre est une base de données, extraire et réutiliser des parties substantielles de l'Oeuvre.

Les droits mentionnés ci-dessus peuvent être exercés sur tous les supports, médias, procédés techniques et formats. Les droits ci-dessus incluent le droit d'effectuer les modifications nécessaires techniquement à l'exercice des droits dans d'autres formats et procédés techniques. L'exercice de tous les droits qui ne sont pas expressément autorisés par l'Offrant ou dont il n'aurait pas la gestion demeure réservé, notamment les mécanismes de gestion collective obligatoire applicables décrits à l'article 4(d).

4. Restrictions. L'autorisation accordée par l'article 3 est expressément assujettie et limitée par le respect des restrictions suivantes :

a. L'Acceptant peut reproduire, distribuer, représenter ou communiquer au public l'Oeuvre y compris par voie numérique uniquement selon les termes de ce Contrat. L'Acceptant doit inclure une copie ou l'adresse Internet (Identifiant Uniforme de Ressource) du présent Contrat à toute reproduction ou enregistrement de l'Oeuvre que l'Acceptant distribue, représente ou communique au public y compris par voie numérique. L'Acceptant ne peut pas offrir ou imposer de conditions d'utilisation de l'Oeuvre qui altèrent ou restreignent les termes du présent Contrat ou l'exercice des droits qui y sont accordés au bénéficiaire. L'Acceptant ne peut pas céder de droits sur l'Oeuvre. L'Acceptant doit conserver intactes toutes les informations qui renvoient à ce Contrat et à l'exonération de responsabilité. L'Acceptant ne peut pas reproduire, distribuer, représenter ou communiquer au public l'Oeuvre, y compris par voie numérique, en utilisant une mesure technique de contrôle d'accès ou de contrôle d'utilisation qui serait contradictoire avec les termes de cet Accord contractuel. Les mentions ci-dessus s'appliquent à l'Oeuvre telle qu'incorporée dans une Oeuvre dite Collective, mais, en dehors de l'Oeuvre en elle-même, ne soumettent pas l'Oeuvre dite Collective, aux termes du présent Contrat. Si l'Acceptant crée une Oeuvre dite Collective, à la demande de tout Offrant, il devra, dans la mesure du possible, retirer de l'Oeuvre dite Collective toute référence au dit Offrant, comme demandé. Si l'Acceptant crée une Oeuvre dite Collective, à la demande de tout Auteur, il devra, dans la mesure du possible, retirer de l'Oeuvre dite Collective toute référence au dit Auteur, comme demandé.

b. L'Acceptant ne peut exercer aucun des droits conférés par l'article 3 avec l'intention ou l'objectif d'obtenir un profit commercial ou une compensation financière personnelle. L'échange de l'Oeuvre avec d'autres Oeuvres protégées par le droit de la propriété littéraire et artistique par le partage électronique de fichiers, ou par tout autre moyen, n'est pas considéré comme un échange avec l'intention ou l'objectif d'un profit commercial ou d'une compensation financière personnelle, dans la mesure où aucun paiement ou compensation financière n'intervient en relation avec l'échange d'Oeuvres protégées.
c. Si l'Acceptant reproduit, distribue, représente ou communique l'Oeuvre au public, y compris par voie numérique, il doit conserver intactes toutes les informations sur le régime des droits et en attribuer la paternité à l'Auteur Original, de manière raisonnable au regard au médium ou au moyen utilisé. Il doit communiquer le nom de l'Auteur Original ou son éventuel pseudonyme s'il est indiqué ; le titre de l'Oeuvre Originale s'il est indiqué ; dans la mesure du possible, l'adresse Internet ou Identifiant Uniforme de Ressource (URI), s'il existe, spécifié par l'Offrant comme associé à l'Oeuvre, à moins que cette adresse ne renvoie pas aux informations légales (paternité et conditions d'utilisation de l'Oeuvre). Ces obligations d'attribution de paternité doivent être exécutées de manière raisonnable. Cependant, dans le cas d'une Oeuvre dite Collective, ces informations doivent, au minimum, apparaître à la place et de manière aussi visible que celles à laquelle apparaissent les informations de même nature.
d. Dans le cas où une utilisation de l'Oeuvre serait soumise à un régime légal de gestion collective obligatoire, l'Offrant se réserve le droit exclusif de collecter ces redevances par l'intermédiaire de la société de perception et de répartition des droits compétente. Sont notamment concernés la radiodiffusion et la communication dans un lieu public de phonogrammes publiés à des fins de commerce, certains cas de retransmission par câble et satellite, la copie privée d'Oeuvres fixées sur phonogrammes ou vidéogrammes, la reproduction par reprographie.

5. Garantie et exonération de responsabilité

a. En mettant l'Oeuvre à la disposition du public selon les termes de ce Contrat, l'Offrant déclare de bonne foi qu'à sa connaissance et dans les limites d'une enquête raisonnable :
    i. L'Offrant a obtenu tous les droits sur l'Oeuvre nécessaires pour pouvoir autoriser l'exercice des droits accordés par le présent Contrat, et permettre la jouissance paisible et l'exercice licite de ces droits, ceci sans que l'Acceptant n'ait aucune obligation de verser de rémunération ou tout autre paiement ou droits, dans la limite des mécanismes de gestion collective obligatoire applicables décrits à l'article 4(e);
b. L'Oeuvre n'est constitutive ni d'une violation des droits de tiers, notamment du droit de la propriété littéraire et artistique, du droit des marques, du droit de l'information, du droit civil ou de tout autre droit, ni de diffamation, de violation de la vie privée ou de tout autre préjudice délictuel à l'égard de toute tierce partie.
c. A l'exception des situations expressément mentionnées dans le présent Contrat ou dans un autre accord écrit, ou exigées par la loi applicable, l'Oeuvre est mise à disposition en l'état sans garantie d'aucune sorte, qu'elle soit expresse ou tacite, y compris à l'égard du contenu ou de l'exactitude de l'Oeuvre.

6. Limitation de responsabilité. A l'exception des garanties d'ordre public imposées par la loi applicable et des réparations imposées par le régime de la responsabilité vis-à-vis d'un tiers en raison de la violation des garanties prévues par l'article 5 du présent contrat, l'Offrant ne sera en aucun cas tenu responsable vis-à-vis de l'Acceptant, sur la base d'aucune théorie légale ni en raison d'aucun préjudice direct, indirect, matériel ou moral, résultant de l'exécution du présent Contrat ou de l'utilisation de l'Oeuvre, y compris dans l'hypothèse où l'Offrant avait connaissance de la possible existence d'un tel préjudice.

7. Résiliation

a. Tout manquement aux termes du contrat par l'Acceptant entraîne la résiliation automatique du Contrat et la fin des droits qui en découlent. Cependant, le contrat conserve ses effets envers les personnes physiques ou morales qui ont reçu de la part de l'Acceptant, en exécution du présent contrat, la mise à disposition d'Oeuvres dites Dérivées, ou d'Oeuvres dites Collectives, ceci tant qu'elles respectent pleinement leurs obligations. Les sections 1, 2, 5, 6 et 7 du contrat continuent à s'appliquer après la résiliation de celui-ci.
b. Dans les limites indiquées ci-dessus, le présent Contrat s'applique pendant toute la durée de protection de l'Oeuvre selon le droit applicable. Néanmoins, l'Offrant se réserve à tout moment le droit d'exploiter l'Oeuvre sous des conditions contractuelles différentes, ou d'en cesser la diffusion; cependant, le recours à cette option ne doit pas conduire à retirer les effets du présent Contrat (ou de tout contrat qui a été ou doit être accordé selon les termes de ce Contrat), et ce Contrat continuera à s'appliquer dans tous ses effets jusqu'à ce que sa résiliation intervienne dans les conditions décrites ci-dessus.

8. Divers

a. A chaque reproduction ou communication au public par voie numérique de l'Oeuvre ou d'une Oeuvre dite Collective par l'Acceptant, l'Offrant propose au bénéficiaire une offre de mise à disposition de l'Oeuvre dans des termes et conditions identiques à ceux accordés à la partie Acceptante dans le présent Contrat.
b. La nullité ou l'inapplicabilité d'une quelconque disposition de ce Contrat au regard de la loi applicable n'affecte pas celle des autres dispositions qui resteront pleinement valides et applicables. Sans action additionnelle par les parties à cet accord, lesdites dispositions devront être interprétées dans la mesure minimum nécessaire à leur validité et leur applicabilité.
c. Aucune limite, renonciation ou modification des termes ou dispositions du présent Contrat ne pourra être acceptée sans le consentement écrit et signé de la partie compétente.
d. Ce Contrat constitue le seul accord entre les parties à propos de l'Oeuvre mise ici à disposition. Il n'existe aucun élément annexe, accord supplémentaire ou mandat portant sur cette Oeuvre en dehors des éléments mentionnés ici. L'Offrant ne sera tenu par aucune disposition supplémentaire qui pourrait apparaître dans une quelconque communication en provenance de l'Acceptant. Ce Contrat ne peut être modifié sans l'accord mutuel écrit de l'Offrant et de l'Acceptant.
e. Le droit applicable est le droit français.

Creative Commons n'est pas partie à ce Contrat et n'offre aucune forme de garantie relative à l'Oeuvre. Creative Commons décline toute responsabilité à l'égard de l'Acceptant ou de toute autre partie, quel que soit le fondement légal de cette responsabilité et quel que soit le préjudice subi, direct, indirect, matériel ou moral, qui surviendrait en rapport avec le présent Contrat. Cependant, si Creative Commons s'est expressément identifié comme Offrant pour mettre une Oeuvre à disposition selon les termes de ce Contrat, Creative Commons jouira de tous les droits et obligations d'un Offrant.

A l'exception des fins limitées à informer le public que l'Oeuvre est mise à disposition sous CPCC, aucune des parties n'utilisera la marque « Creative Commons » ou toute autre indication ou logo afférent sans le consentement préalable écrit de Creative Commons. Toute utilisation autorisée devra être effectuée en conformité avec les lignes directrices de Creative Commons à jour au moment de l'utilisation, telles qu'elles sont disponibles sur son site Internet ou sur simple demande.

Creative Commons peut être contacté à http://creativecommons.org/.