Un tournoi de programmes de Phutball

Tristan Cazenave
Labo IA, Dept Informatique, Université Paris 8,
2 Rue de la Liberté, 93526 Saint-Denis, France.
cazenave@ai.univ-paris8.fr

Résumé: Pour mieux comprendre comment on découvre des concepts qui permettent à un programme de bien résoudre des problèmes complexes, nous avons tenté une expérience collective visant à écrire des programmes pour un jeu complexe inconnu de tous les participants. Chaque programme a été écrit par une équipe de quatre personnes. Chaque équipe contenait une personne chargée de noter les idées qui étaient exprimées. Le jeu concerné était le Football des philosophes ou Phutball.

Mots-clés: Jeu, Tournoi, Programme, Phutball, Découverte de concepts, Monitoring.

1. Introduction

Pour mieux comprendre comment on découvre des concepts qui permettent à un programme de bien résoudre des problèmes complexes, nous avons tenté une expérience collective visant à écrire des programmes pour un jeu complexe inconnu de tous les participants. Chaque programme a été écrit par une équipe de quatre personnes. Chaque équipe contenait une personne chargée de noter les idées qui étaient exprimées. Le jeu concerné était le Football des philosophes ou Phutball.

Le Football des philosophes est décrit dans Winning Ways [Berlekamp 1982], un ouvrage sur la théorie combinatoire des jeux [Conway 1976] appliquée à de nombreux jeux. Il a été surnommé Phutball par J. H. Conway et les auteurs de Winning Ways pensent que ce jeu ne peut pas être totalement analysé par la théorie combinatoire des jeux. Je pense au contraire que ce jeu a de bonnes chances d'être résolu en utilisant des techniques de recherches appropriées et de la démonstration de théorèmes tactique dans le style d'Introspect [Cazenave 1998], mais cela reste encore a faire…

Nous commençons par décrire les règles du jeu, puis nous explicitons les contraintes et les programmes qui ont été donnés à chacune des équipes [Cazenave 1999]. Nous décrivons ensuite les résultats de chaque équipe. Puis nous donnons certains concepts intéressants mis a jour par l'équipe Socrate. Nous tentons enfin de conclure en analysant les résultats cette expérience.

2. Règles du jeu

Le Football des philosophes est joué sur un damier de Go de 19x19, une pierre noire représente la balle et les pierres blanches représentent les joueurs de Football. Toutes les pièces sont communes aux deux joueurs, et les deux joueurs ont les mêmes coups légaux.

La partie commence avec un damier vide, la balle est placée sur l'intersection centrale. Ensuite, à chaque coup, chaque joueur doit :

- soit poser une nouvelle pierre blanche sur une intersection vide
- soit faire sauter la balle par dessus des pierres blanches, en enlevant les pierres sautées.

Un saut peut être dans n'importe laquelle des 8 directions.
Le but du jeu est de faire parvenir la balle sur la première ligne du camp adverse, ou derrière cette première ligne.

3. Contraintes sur les Joueurs informatiques

1) Chaque joueur dispose de 5 minutes pour jouer une partie. Le premier joueur qui dépasse 5 minutes de jeu a perdu.
2) Une partie est déclarée nulle, si au bout de 200 coups aucun des joueurs n'a gagné ni n'a dépassé son temps imparti.
3) Une partie gagnée vaut 2 points, une partie nulle 1 point, et une partie perdue 0 points.
4) Chaque joueur a un nom de philosophe parmi : Socrate, Kant, Platon, Descartes, LaoTseu. Les fichiers contenant les fonctions à écrire sont les fichiers commençant par le nom du philosophe et suffixée par .cpp (fonctions c++ ou c) et .h (en-têtes).
5) L'algorithme utilisé pour choisir est un Alpha-Béta simple. Chaque joueur peut régler la profondeur en utilisant la variable définie au début du fichier .cpp, par exemple int ProfondeurMaxSocrate=5; pour Socrate dans Socrate.cpp.

6) La valeur Infini est définie par 1 000 000, le fonctions d'évaluation doivent donc renvoyer des entiers entre –1 000 000 et 1 000 000.

4. Fonctions Prédéfinies

Le damier est un damier de Go de 19 intersections sur 19 intersections. Il est représenté par un tableau de 23 entiers non signés :

```
unsigned int DamierPrincipal [23] ;
```

Les fonctions suivantes sont disponibles :

```
Vide(damier,i)      => indique si l'intersection i du damier est vide
Couleur(damier,i)   => renvoie la couleur de l'intersection i parmi AMI, ENNEMI,
VIDE
IntersectionBalle(damier) => renvoie l'intersection sur laquelle se trouve la
balle, les intersections varie de 0 à 360, on a en plus deux intersections
particulières définies par les constantes BUT_GAUCHE et BUT_DROIT
BordGauche(i)       => renvoie 1 si la balle est sur le bord gauche
BordDroit (i)       => renvoie 1 si la balle est sur le bord droit
BordHaut (i)        => renvoie 1 si la balle est sur le bord haut
BordBas (i)         => renvoie 1 si la balle est sur le bord bas
NumeroIntersection(x,y)   => renvoie l'intersection de coordonnées x,y
Abscisse(i)         => Abscisse d'une intersection
Ordonnee (i)        => Ordonnée d'une intersection
```

On dispose aussi des trois tableaux :

```
int TableauVoisines[361][5]; int TableauDiagonales[361][5]; et
int TableauVoisinesEtDiagonales[361][9];
```

Si on veut parcourir toutes les Voisines et Diagonales d'une intersection Inter on utilise la boucle :

```
int InterProche ;
for (i=1; i<TableauVoisinesEtDiagonales[Inter][0]+1 ; i++)
   {
     InterProche=TableauVoisinesEtDiagonales[Inter][i] ;
   }
```

5. Exemple de Programme

Le programme suivant correspondant au joueur Socrate donne une idée de l'utilisation possible des primitives du jeu. La fonction à écrire pour chaque philosophe est la fonction :

```
int EvaluationSocrate(uint *Position)
```

qui renvoie l'évaluation d'une position, on peut aussi modifier la fonction qui envisage les coups possibles :

```
void TrouveCoupsPossiblesSocrate(uint *Position,int &NombrePositionsSuivantes,uint
PositionSuivante[NOMBRE_COUPS_MAX][23])
```

Voici ce que donne le programme initial de chaque philosophe :

```
#define SOCRATE_C

#include <stdio.h>
#include <iostream.h>

#include "damier.h"
#include "search.h"
#include "Socrate.h"

int ProfondeurMaxSocrate=4;
int NombreNoeudsMaxSocrate=100000;

static int TableauIntersectionsAtteignables [500];

void TrouveCoupsPossiblesSocrate(uint *Position,int &NombrePositionsSuivantes,uint
PositionSuivante[NOMBRE_COUPS_MAX][23])
  {
   int i,j,balle=IntersectionBalle(Position),inter;
   // commencer par les deplacement de la balle parce que ce sont
   // souvent des coups qui rapportent des points
   TableauIntersectionsAtteignables[0]=0;
   AjouteElementTableau(TableauIntersectionsAtteignables,balle);
   CoupsPossiblesBalleSocrate(Position,NombrePositionsSuivantes,PositionSuivante);
   for (j=1; j<TableauIntersectionsAtteignables[0]+1; j++)
      {
       inter=TableauIntersectionsAtteignables[j];
       if ((inter>=0)&&(inter<NombreIntersections))
       if (Couleur(Position,inter)==VIDE)
         {
          AffecteDamier(PositionSuivante[NombrePositionsSuivantes],Position);
          Joue(PositionSuivante[NombrePositionsSuivantes],inter,PION);
          NombrePositionsSuivantes++;
         }
       if ((inter>=0)&&(inter<NombreIntersections))
       for (i=1; i<TableauVoisinesEtDiagonales[inter][0]+1; i++)
       if (Couleur(Position,TableauVoisinesEtDiagonales[inter][i])==VIDE)
         {
          AffecteDamier(PositionSuivante[NombrePositionsSuivantes],Position);

Joue(PositionSuivante[NombrePositionsSuivantes],TableauVoisinesEtDiagonales[inter]
[i],PION);
          NombrePositionsSuivantes++;
         }
      }
   if (NombrePositionsSuivantes>=NOMBRE_COUPS_MAX)
     cout << "debug";
  }

static int DeltaPossibles[]={1,-1,19,-19,20,-20,18,-18};
void CoupsPossiblesBalleSocrate (uint *Position,int &NombrePositionsSuivantes,uint
PositionSuivante[NOMBRE_COUPS_MAX][23])
  {
   int i,j,Delta,inter1;
   uint *PositionTemporaire;
   for (i=0; i<8; i++)
      {
       Delta=DeltaPossibles[i];
       inter1=IntersectionBalle(Position)+Delta;
       if ((inter1>-1)&&(inter1<NombreIntersections))
       if (Couleur(Position,inter1)==PION)
```

```
         {
          while (!Bord(inter1)&&(inter1>-
1)&&(inter1<NombreIntersections)&&(Couleur(Position,inter1)==PION))
            inter1+=Delta;
          // on est forcement a l'interieur du damier
          if (Couleur(Position,inter1)==VIDE)
            {
             // dupliquer la position
             PositionTemporaire=PositionSuivante[NombrePositionsSuivantes];
             AffecteDamier(PositionTemporaire,Position);
             NombrePositionsSuivantes++;
             // jouer le coup et oter les pions sautes
             JoueVide(PositionTemporaire,IntersectionBalle(PositionTemporaire));
             JoueVide(PositionTemporaire,inter1);
             for (j=IntersectionBalle(PositionTemporaire)+Delta; j!=inter1;
j+=Delta)
                JoueVide(PositionTemporaire,j);
             Joue(PositionTemporaire,inter1,BALLE);
             AffecteBalle(PositionTemporaire,inter1);
             AjouteElementTableau(TableauIntersectionsAtteignables,inter1);
             // appel recursif aux coups suivants possibles

CoupsPossiblesBalleSocrate(PositionTemporaire,NombrePositionsSuivantes,PositionSui
vante);
            }
          // inter1 n'est pas vide et on est donc au bord
          else if (BordGauche(inter1)||BordDroit(inter1))
            {
             // dupliquer la position
             PositionTemporaire=PositionSuivante[NombrePositionsSuivantes];
             AffecteDamier(PositionTemporaire,Position);
             NombrePositionsSuivantes++;
             // jouer le coup et oter les pions sautes
             JoueVide(PositionTemporaire,IntersectionBalle(PositionTemporaire));
             JoueVide(PositionTemporaire,inter1);
             for (j=IntersectionBalle(PositionTemporaire)+Delta; j!=inter1;
j+=Delta)
                JoueVide(PositionTemporaire,j);
             if (BordGauche(inter1))
               AffecteBalle(PositionTemporaire,BUTGAUCHE);
             else if (BordDroit(inter1))
               AffecteBalle(PositionTemporaire,BUTDROIT);
            }
         }
      }
  }

int EvaluationSocrate(uint *Position)
  {
   int Eval;
   if (IntersectionBalle(Position)==BUTDROIT)
     Eval=INFINI;
   else if (IntersectionBalle(Position)==BUTGAUCHE)
     Eval=-INFINI;
   else
     Eval=Abscisse(IntersectionBalle(Position));
   if (JoueurAmi==GAUCHE)
     return Eval;
   else
     return -Eval;
  }
```

6. Les différentes équipes

Platon

Platon est le meilleur programme de Phutball de cette compétition. La fonction d'évaluation de Platon est aussi la plus simple, elle ne renvoie que trois valeurs, -INFINI lorsque la partie est perdue, +INFINI lorsqu'elle est gagnée, 0 autrement :

```
int ProfondeurMaxPlaton=6;

int EvaluationPlaton(uint *Position)
  {
   int Eval;
   if ((IntersectionBalle(Position)==BUTDROIT)||(Abscisse(IntersectionBalle(Position))==19))
     Eval=INFINI;
   else if ((IntersectionBalle(Position)==BUTGAUCHE)||(Abscisse(IntersectionBalle(Position))==1))
     Eval=-INFINI;
   else
     Eval=0;
   if (JoueurAmi==GAUCHE)
     return Eval;
   else
     return -Eval;
}
```

Cette fonction d'évaluation simpliste permet en fait de trouver un gain ou d'empêcher une perte que l'on peut prévoir 6 coups à l'avance. Elle n'est pas utilisée pour choisir les coups si un gain n'est pas en vue. L'approche de Platon est de donner directement le meilleur coup à jouer. C'est le coup qui se trouve à un saut de la pierre la plus proche du bord adverse :

```
extern int ProfondeurCourante;
void TrouveCoupsPossiblesPlaton(uint *Position,int &NombrePositionsSuivantes,uint
PositionSuivante[NOMBRE_COUPS_MAX][23])
  {
   int i,j,balle=IntersectionBalle(Position),inter;
   int trouve=0;
   // commencer par les deplacement de la balle parce que ce sont
   // souvent des coups qui rapportent des points
   if (Even(ProfondeurCourante)) {
   if (JoueurAmi==GAUCHE) {
     if (Vide(Position,balle+1))  {
       trouve=1;
       AffecteDamier(PositionSuivante[NombrePositionsSuivantes],Position);
       Joue(PositionSuivante[NombrePositionsSuivantes],balle+1,PION);
       NombrePositionsSuivantes++;  }
     for (inter=balle+1; ((inter<361)&&!trouve); inter+=1)
     if (inter<361)
     if (Vide(Position,inter)) {
       if (Abscisse(inter)<19) {
         if (Vide(Position,inter+1)) {
           trouve=1;
           AffecteDamier(PositionSuivante[NombrePositionsSuivantes],Position);
           Joue(PositionSuivante[NombrePositionsSuivantes],inter+1,PION);
           NombrePositionsSuivantes++; }}}}
   else{
     if (Vide(Position,balle-1)){
       trouve=1;
       AffecteDamier(PositionSuivante[NombrePositionsSuivantes],Position);
       Joue(PositionSuivante[NombrePositionsSuivantes],balle-1,PION);
```

```
      NombrePositionsSuivantes++;}
    for (inter=balle-1; ((inter<361)&&!trouve); inter-=1)
    if (inter>=0)
    if (Vide(Position,inter)){
      if (Abscisse(inter)>1){
        if (Vide(Position,inter-1)){
          trouve=1;
          AffecteDamier(PositionSuivante[NombrePositionsSuivantes],Position);
          Joue(PositionSuivante[NombrePositionsSuivantes],inter-1,PION);
          NombrePositionsSuivantes++;}}}}

  TableauIntersectionsAtteignables[0]=0;
  AjouteElementTableau(TableauIntersectionsAtteignables,balle);
  CoupsPossiblesBallePlaton(Position,NombrePositionsSuivantes,PositionSuivante);
  }
  else  {
  TableauIntersectionsAtteignables[0]=0;
  AjouteElementTableau(TableauIntersectionsAtteignables,balle);
  CoupsPossiblesBallePlaton(Position,NombrePositionsSuivantes,PositionSuivante);
  for (j=1; j<TableauIntersectionsAtteignables[0]+1; j++)  {
     inter=TableauIntersectionsAtteignables[j];
     if ((inter>=0)&&(inter<NombreIntersections))
     for (i=1; i<TableauVoisinesEtDiagonales[inter][0]+1; i++)
     if (Vide(Position,TableauVoisinesEtDiagonales[inter][i])){
       AffecteDamier(PositionSuivante[NombrePositionsSuivantes],Position);
       Joue(PositionSuivante[NombrePositionsSuivantes],TableauVoisinesEtDiagonales[inter][i],PION);
       NombrePositionsSuivantes++;}} } }
```

Pour arriver à ce résultat l'équipe de Platon est passé par les états suivants :

Elle a commencé par jouer contre les programmes qui étaient fournis, elle fait tout de suite deux constatations : il vaut mieux que les pions soient espacés d'une intersection plutôt que d'être contigus, et on ne peut pas bloquer l'adversaire sauf si on lui court-circuite la chaîne la plus longue vers son but. Les questions qui se posent tout de suite sont : le minimax est il utile ? La profondeur de l'Alpha-Béta peut-elle être modifiée en fonction du temps restant ?

L'équipe joue ensuite avec l'ordinateur pour trouver des coups intéressants : après des sauts successifs, il est bon de manger le maximum de pierres autour de la balle tout en restant près du but, il faut empêcher l'adversaire d'arriver à l'avant-avant dernière ligne verticale…

La première idée qui vient pour évaluer une position est de maximiser la valeur nb(son camp)-nb(notre camp). Ensuite l'équipe joue avec Simplet et se rend compte qu'il les aide à gagner ! Elle décide alors de mettre l'Alpha-Béta à une petite profondeur car elle sait ou jouer : on joue toujours entre la balle est le but adverse et le plus près possible de la balle. Elle teste alors cette stratégie dans une partie humain-humain, elle note alors que dans cette situation :

```
+ + O + O + O @ O + O + O + O
```

il faut manger pour se retrouver dans cette situation ou Droit ne peut plus empêcher Gauche de gagner :

```
+ @ + + + + + + O + O + O + O
```

Le temps pressant, l'équipe passe alors à la programmation. Le premier essai consiste à faire intervenir la balance des pions de façon très simple, et cela donne un mauvais programme !

La fonction d'évaluation est alors changée : les pions de notre coté compte –10 et ceux du coté de l'adversaire compte +10. La fonction d'évaluation donne la même valeur pour un coup que l'équipe souhaitait et pour un coup que l'équipe ne souhaitait pas. Au lieu de jouer du coté

de son but, il préfère manger vers son camp ! Un test avec +200/-100 ne marche pas non plus. L'équipe essaie ensuite une balance pondérée et de compter les bons et les mauvais pions mais ca ne marche pas.
L'équipe décide alors d'abandonner cette approche et de faire ce qui les tentaient depuis longtemps : programmer le seul coup qui leur semble bon à chaque fois. Ils se rendent compte qu'ils connaissent le bon coup à jouer dans chaque situation mais qu'il est difficile de le faire trouver par une fonction d'évaluation numérique. La stratégie est la suivante : nous sommes dans le camp droit, nous voulons atteindre le bord gauche. Si la position immédiatement à gauche de la balle est vide : placer le joueur, sinon on cherche l'extrémité horizontale vers le but, on passe une intersection vide et on place le jouer sur l'intersection vide suivante. Après des test, l'équipe se rend compte qu'elle ne peut pas se passer de l'Alpha-Béta à cause des coups extrêmes (ceux qui sont près du bord et qui permettent le gain). Ils remettent alors un Alpha-Béta extrêmement simpliste. Toutefois lorsque le programme envisage les coups de l'adversaire dans l'Alpha-Béta, il ne considère que le coup obligatoire ce qui l'empêche parfois de voir une parade de l'adversaire ou d'empêcher l'adversaire de gagner. Le programme est alors modifié pour que tous les coups de l'adversaire soient pris en compte. La programmation n'est pas terminée pour la compétition du soir, mais le programme est terminé plus tard dans la nuit, il contient alors correctement les idées énoncées auparavant. Platon gagne alors contre tous ses adversaires, dans le camp gauche comme dans le camp droit !

Lao-Tseu

Lao Tseu est arrivé deuxième du tournoi, lui aussi avec une fonction d'évaluation assez simple et sans recherche arborescente !

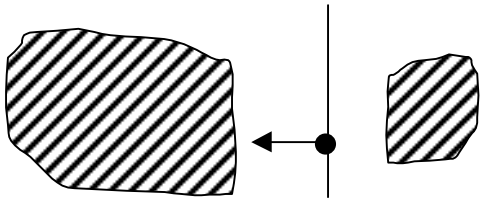
Voici la fonction d'évaluation de Lao-Tseu :

```
int ProfondeurMaxLaoTseu=1;

int EvaluationLaoTseu(uint *Position)
  {
   int Eval = 0;
   if(IntersectionBalle(Position)==BUTDROIT)
     Eval=INFINI;
   else if(IntersectionBalle(Position)==BUTGAUCHE)
     Eval=-INFINI;
   else {

     int abc = Abscisse(IntersectionBalle(Position));
     int ord = Ordonnee(IntersectionBalle(Position));


     if(JoueurAmi==GAUCHE) {
       Eval = abc;
       for(int i = abc+1; i <= 19; i++) {
          int inters = NumeroIntersection(i,ord);
          int j = i - abc;
          if((j % 2 == 1) && (Couleur(Position,inters) == PION)) {
             Eval += 19 - j;
          }//end if(Couleur(Position,i) == PION)
       }//end for i
       return(Eval);
```

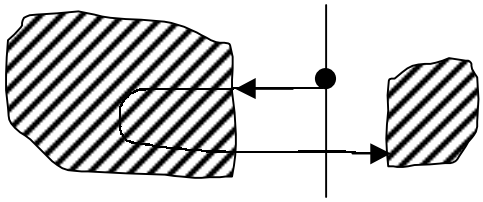
```
    }//end if(JoueurAmi==GAUCHE)

   if(JoueurAmi==DROITE) {
     Eval = 19-abc;
     for(int i = 1; i < abc; i++) {
        int inters = NumeroIntersection(i,ord);
        int j = abc - i;
        if((j % 2 == 1) && (Couleur(Position,inters) == PION)) {
           Eval += 19 - j;
        }//end if(Couleur(Position,i) == PION)
     }//end for i
     return(Eval);
    }//end if(JoueurAmi==DROITE)

 }//end else

 if (JoueurAmi==GAUCHE)
   return (Eval);
 else
   return (-Eval);

}
```

Voici comment elle a été élaborée :

L'équipe a commencé par rechercher des caractéristiques de la fonction d'évaluation et les poids associés à chaque coefficient. La première idée qui leur est venue a été de compter le nombre de pierres dans chaque camp :



L'équipe a ensuite pensé qu'il était plus urgent de se déplacer que de poser des pierres. En comparant aux dames chinoises, l'équipe a ensuite noté que les pions sautés étaient enlevé et que le retour arrière était donc plus difficile. Une autre idée apparue assez tôt fut que l'on devait comptabiliser le nombre de coups légaux à partir de la position de la balle. L'équipe a ensuite décidé de faire des parties pour mieux se rendre compte de ce qui était important. En jouant elle s'est rendue compte qu'il était intéressant de poser les pierres toutes les deux intersections, et que les cases paires par rapport à la direction de la balle devaient être privilégiées. L'équipe a ensuite pensé de nouveau qu'il était intéressant de se déplacer le plus vite possible dans la direction du but (dès que plus de deux pierres sont alignées dans cette direction). Une question est alors apparue : faut-il privilégier les parcours qui font passer chez l'adversaire pour lui prendre des pierres :



L'équipe est ensuite revenu de cette assertion en se rendant compte qu'en début de partie, il fallait privilégier la pose des pierres par rapport au déplacement. Toutefois, si un coup amène au but de l'adversaire, il faut le jouer. L'équipe s'est alors intéressée aux stratégies de blocage : on ne peut pas empêcher un déplacement en ligne droite, mais on peut empêcher un déplacement en zigzag. Trois stratégies différentes ont donc été mises à jour : Blocage, Développement et Déplacement. Tout cela étant resté très théorique jusqu'ici, l'équipe a décidé de commencer par écrire une fonction d'évaluation qui ne prend en compte que le nombre de pierres de chaque coté de la balle. En testant cette fonction d'évaluation, les programmeurs se rendent compte que si le programme joue Gauche, il déplace la balle vers son camp de façon à avoir le maximum de pierres à sa droite, et que s'il part vers la droite, il remet des pierres dans la partie gauche et joue alors contre son camp !
L'équipe décide alors de réfléchir aux bons coups de déplacement et aux bons coup pour poser une pierre. Elle décide alors d'enlever de la fonction d'évaluation le nombre de pierres, et plutôt de prendre en compte la position des pierres les unes par rapport aux autres en cherchant des patterns. L'accent est alors mis sur la recherche de chaînes qui permettent d'avancer. Après plusieurs idées et essais infructueux, le programme ne faisant jamais ce qu'on pensait qu'il ferait avec la stratégie implémentée, l'équipe est découragée. Elle joue alors contre le programme fournit au départ et trouve une stratégie très simple pour gagner contre lui : à chaque fois qu'il s'avance, remettre une pierre derrière lui pour s'accrocher !
Ils se rendent compte alors qu'il arrivent bien à exprimer une stratégie mais qu'ils n'arrivent pas à la programmer comme une fonction d'évaluation. Ils décident donc de ne plus utiliser l'alpha-béta et de définir directement une stratégie à profondeur 1 : si pas de pierre à droite de la balle, on augmente au maximum pour qu'elle aille vers la droite et ne cherche pas à reculer. Cette stratégie marche quand on joue en premier mais pas quand on joue en deuxième !

Socrate

Voici la fonction d'évaluation de Socrate :

```
int ProfondeurMaxSocrate=4;

int EvaluationSocrate(uint *Position)
  {
   int Eval;
   if (IntersectionBalle(Position)==BUTDROIT)
     Eval=INFINI;
   else if (IntersectionBalle(Position)==BUTGAUCHE)
     Eval=-INFINI;
```

```
  else
     {
      //if (JoueurAmi==GAUCHE)

Eval=AbscisseMaxAtteignableSocrate(Position)+AbscisseMinAtteignableSocrate(Positio
n);
      //else
      //  Eval=AbscisseMinAtteignable(Position)-AbscisseMaxAtteignable(Position);

      Eval=Eval;//*100+Abscisse(IntersectionBalle(Position));
     }
     /*else
     {
      if (JoueurAmi==GAUCHE)
        Eval=AbscisseMaxAtteignableSocrate(Position);
      else
        Eval=AbscisseMinAtteignableSocrate(Position);

      Eval+=0*Abscisse(IntersectionBalle(Position));
     }
        */
   if (JoueurAmi==GAUCHE)
     return Eval;
   else
     return -Eval;
  }
```

Pour commencer, l'équipe de Socrate joue avec l'ordinateur et sur un damier. Elle découvre alors plusieurs concepts: l'importance du retour, il est intéressant d'avoir des pions 'couvrants' entre la balle et le but (compter le nombre de pions entre la balle peut être intéressant), elle suppose ensuite que la fonction dépendra fortement de la distance de la balle au but (cette idée sera complètement remise en cause plus tard). L'équipe se scinde alors en deux sous-groupes : un sous-groupe qui cherche à piéger l'adversaire et qui critique fortement le principe de l'Alpha-Béta, ce premier sous-groupe réfléchit sur la mémorisation des coups de l'adversaire et pense à copier ce qu'il fait, il pense ajuster des paramètres de la fonction d'évaluation en fonction de l'historique, il propose vers la fin de mettre au point un paramètre d'attaque qui augmente si on n'a pas été en danger au cours des dix derniers coups. Le deuxième groupe a une approche plus pragmatique et accepte d'utiliser l'Alpha-Béta. Il cherche donc à écrire une fonction d'évaluation simple sachant qu'il a peu de temps pour le faire. Il joue alors avec l'ordinateur et examine la fonction d'évaluation du programme fourni le plus fort. Elle essaie de régler des paramètres de cette fonction d'évaluation, mais ne pense pas toujours à optimiser le jeu à la fois quand on joue en premier et quand on joue en deuxième. Vient une idée : faire intervenir le max de la position atteignable par l'adversaire, elle utilise alors dans la fonction d'évaluation l'expression max-min+a*pos-balle. Elle se rend compte alors sur un exemple et après réflexion que l'expression qui convient à leur idée est en fait max+min+a*pos-balle. Après des tests, elle se rend compte que a=0 donne la meilleure fonction d'évaluation et bat le programme fourni le plus fort aussi bien en premier qu'en deuxième. Elle essaie alors de mettre un coefficient sur max, mais les essais ne sont pas concluants.

Kant

Voici la fonction d'évaluation de Kant :

```
int ProfondeurMaxKant=4;

#define SeuilCoupsPossibles 100
#define COEFBALLE               100
#define COEFNBCOUP              5
#define COEFORDONNEE      100
#define COEFPIERRE              20
#define AbscisseRelative(i)     ((JoueurAmi==GAUCHE)? Abscisse(i) : (19 -
Abscisse(i)))

int EvaluationKant(uint *Position)
{   int Sum=0,SumOrd=0,nbpierre=0;
    int n=0,Eval=0,iold,inew,ordold;
    static uint PosSuiv[NOMBRE_COUPS_MAX][23];
     int diffabs;


       /* critere 1 */
     /*iold=IntersectionBalle(Position);
    CoupsPossiblesBalleKant(Position,n,PosSuiv);
     for (int c=0; c<n; c++)
     {  inew=IntersectionBalle(PosSuiv[c]);
       diffabs=AbscisseRelative(inew)-AbscisseRelative(iold);
       Eval+=diffabs;
     }*/

       /* critere 2 */
     if (IntersectionBalle(Position)==BUTDROIT)
     {  if (JoueurAmi==GAUCHE)
              return(INFINI);
       else
              return(-INFINI);
     }
    else if (IntersectionBalle(Position)==BUTGAUCHE)
    {  if (JoueurAmi==GAUCHE)
              return(-INFINI);
       else
              return(INFINI);
     }
    else
       Eval=COEFNBCOUP*Eval + COEFBALLE*AbscisseRelative(iold);

         /* critere 3 */
         ordold=abs(Ordonnee(iold)-10);
         Eval+=COEFORDONNEE*ordold;

       /* critere 4 */
    for (int i=0; i<361; i++)
       if (Couleur(Position,i)==PION)
         {   Sum+=AbscisseRelative(i);
              SumOrd+=abs(abs(Ordonnee(i)-10) - ordold);
              nbpierre++;
         }
    if (nbpierre)
        if (JoueurAmi==GAUCHE)
         if (AbscisseRelative(iold)>10)
              Eval+=(COEFPIERRE*(Sum + SumOrd))/nbpierre;
         else  Eval+=(COEFPIERRE*(Sum - SumOrd))/nbpierre;
        else  Eval+=(COEFPIERRE*(Sum + SumOrd))/nbpierre;

     return Eval;
}
```

L'équipe de Kant n'étant pas extrêmement fier de ses résultats, elle a préférée ne pas s'étendre en détail sur ses idées. Une des idées qu'elle a expérimentée était de maximiser le nombre de coups possible pour compliquer au maximum la situation et peut-être faire perdre l'adversaire

au temps. Toutefois cette stratégie amène à des positions ou plus de 1 000 coups sont possibles sur une seule position, ce qui a fait exploser la pile que j'avais prévue pour stocker toutes les positions qui suivent une position !

Descartes

Voici la fonction d'évaluation de Descartes :

```
int ProfondeurMaxDescartes=4;

static uint PositionsTemporairesDescartes[NOMBRE_COUPS_MAX][23];

int EvaluationDescartes(uint *Position)
  {
   int Eval,entierinutile=0;
   ScoreChemins = 0 ;
   AbscisseBalleInit = Abscisse(IntersectionBalle(Position));
   LongueurCheminsDescartes(0,Position,
entierinutile,PositionsTemporairesDescartes) ;
   if (IntersectionBalle(Position)==BUTDROIT)
     Eval=INFINI;
   else if (IntersectionBalle(Position)==BUTGAUCHE)
     Eval=-INFINI;
   else
     Eval = (Abscisse(IntersectionBalle(Position)))+ScoreChemins ;

   if (JoueurAmi==GAUCHE)
     return Eval;
   else
     return -Eval;
  }
```

L'équipe de Descartes n'a pas réussi a trouver d'idées qui améliore les programmes fournis. Elle a donc utilisé le meilleur programme fourni avec une profondeur incrémentée, cependant le programme résultant était souvent trop lent (temps de réponse parfois de l'ordre de la minute).

Résultats du tournoi

Tous les programmes ont rencontré tous les autres programmes, une fois en étant le joueur Gauche, une fois en étant le joueur Droit. Une partie nulle compte pour un point, et une partie gagnée compte pour deux points. Comme chaque programme a 6 opposants, il fait douze parties, le score maximum est donc de 24. Ce score a été atteint par Platon qui a gagné toutes ses parties avec Gauche et avec Droite. Les deux joueurs déjà programmés donc le code était fourni dès le début aux participants étaient Simplet et Introspect :

Platon	24
Lao-Tseu	16
Socrate	16
Introspect	11
Kant	8
Descartes	8

7. Les concepts mis à jour par la suite

Jean-Yves Lucas de l'équipe Socrate a continué par la suite la mise à jour de concepts du Phutball. Pour réfléchir sur les règles du jeu, il propose de modifier les règles pour voir ce qui change dans le jeu :

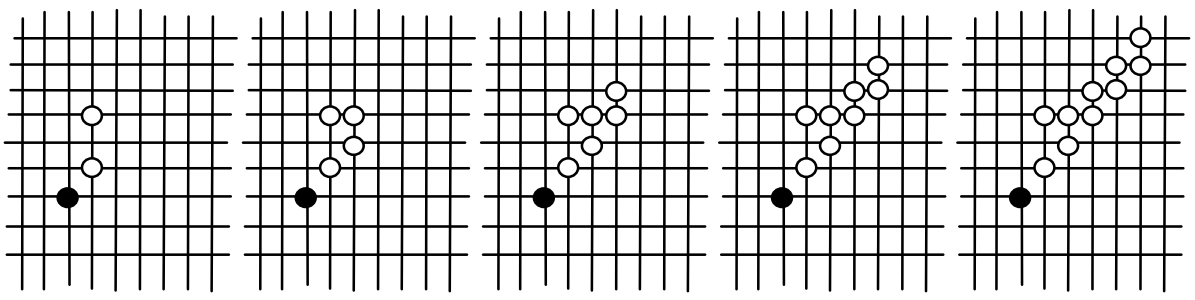
- modifier la taille du tableau de jeu.
- modifier la règle de prise.
- modifier les coups (2 coups, ôter un joueur…).
- modifier le nombre de joueurs.
- modifier l'emplacement initial de la balle.
- placer des joueurs au départ (handicap).

Il propose ensuite d'utiliser le monitoring pour améliorer le jeu, par exemple il propose de détecter par monitoring du programme par lui-même :

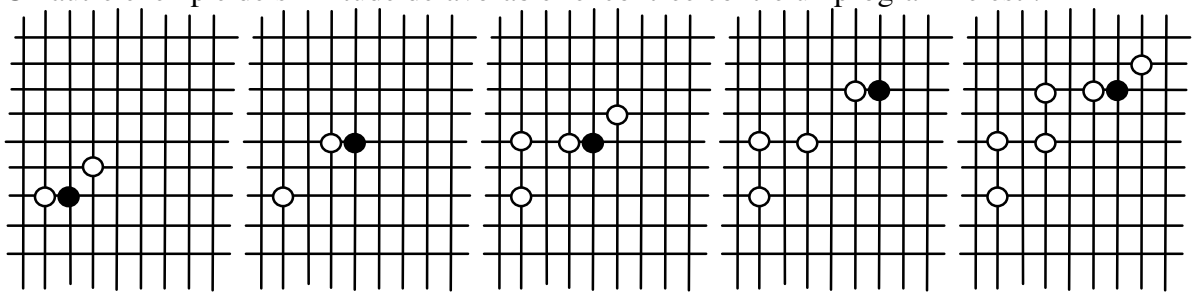
- les similitudes entre situations.
- les situations gagnantes.
- les coups qui permettent de gagner au plus vite.

Il présente alors les concepts de bord et de zone de danger.

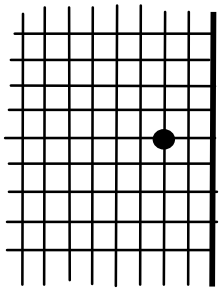
Du point de vue du monitoring du programme par lui-même, il déplore par exemple que le programme ne se rende pas compte qu'à chaque fois qu'il joue il se retrouve dans une situation quasi-identique à la précédente mais de plus en plus défavorable. L'exemple le plus flagrant est Simplet qui mange la pierre qui est posée à coté de la balle vers son camp parce qu'il pense que tous les coups sont équivalents (il joue au niveau 3 et ne peut donc pas empêcher l'adversaire de jouer ce coup) ! Jean Yves Lucas aimerait que le programme détecte qu'il va vers une perte inéluctable s'il ne change pas de stratégie. Un autre exemple est donné par un autre programme de Phutball qui continue à ajouter des pierres en diagonales dans son camp pour se ménager un retour :



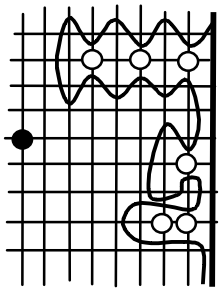
Un autre exemple de similitude défavorable rencontrée contre un programme est :



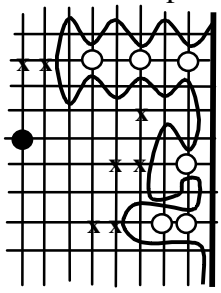
Un exemple de situation gagnantes qui pourrait être déterminée directement sans calcul est quand la balle est isolée sur la troisième ligne. On peut utiliser de la démonstration de théorèmes pour trouver ces situations et les intégrer dans le programme. J'indique au passage qu'Introspect a été utilisé dans ce sens et a commencé à démontrer des théorèmes tactiques du Phutball… Exemple de situation gagnée pour Gauche quelque soit le trait :



Concept de bord : ensemble des intersections qui donnent le gain si elles peuvent être atteintes par la balle. Exemple :



Concept de zone de danger : ensemble des intersections qui permettent une suite de coups gagnants (prés du bord et pas de joueurs alentours pour s'éloigner) :



8. Conclusion

Il apparaît clairement que les meilleurs programmes sont ceux dont les équipes ont su sortir du cadre de l'Alpha-Béta, car dans ce jeu il existe une stratégie simple qui donne tout de suite des résultats, alors que la définition des concepts qui permettent d'évaluer une position est difficile. On peut noter toutefois l'exception de l'équipe Socrate qui a réussi a écrire une fonction d'évaluation simple qui a de bons résultats. Le Phutball semble donc être plus proche du Go que des Echecs, non seulement parce qu'il utilise un damier de Go mais aussi parce que les fonctions d'évaluation des coups marchent mieux que les fonctions d'évaluation des positions (ceci dans le temps très limité qui était donné aux différentes équipes pour écrire leur programme).

La motivation initiale de cette expérience était de mieux comprendre les mécanismes qui nous permettent de découvrir des concepts utiles pour programmer des résolveurs de problèmes complexes. Ce papier est un résumé de la chronologie des différentes idées qui sont apparues et de leur évaluation. C'est une expérience de terrain qui est encore bien loin d'une théorie assez formalisées pour en faire un programme, elle peut tout de même apporter un éclairage sur les méthodes à suivre ou à éviter pour chercher et trouver des heuristiques utilisables dans des programmes.

Le Phutball semble se prêter à l'utilisation de la démonstration de théorèmes tactiques qui permettraient de prévoir le gain de nombreux coups à l'avance, et peut-être de le résoudre, Introspect sera bientôt utilisé dans cette optique. Il reste aussi à tester les concepts introduit par Jean-Yves Lucas et à tester leur efficacité pour mieux jouer et pour accélérer la preuve de positions. J'ai fait une page web du Phutball pour toutes les personnes désireuses de participer et de continuer le tournoi [Cazenave 1999].

9. Bibliographie

[Berlekamp 1982] - E. Berlekamp, J.H. Conway, R.K. Guy. *Winning Ways*. Academic Press, London 1982.

[Cazenave 1999] – T. Cazenave, *http://www.ai.univ-paris8.fr/~cazenave/computergames.html*

[Cazenave 1998] – T. Cazenave, *Metaprogramming Forced Moves.* Proceedings ECAI98,Brighton, 1998.

[Conway 1976] - J. Conway, *On Numbers and Games,* Academic Press, Londres/New-York, 1976.