République Algérienne Démocratique et Populaire
الجمهوريـــة الجـزائريـة الديمـقراطيـة الشـعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التـعليـم العـالي و البـحث العلـمي

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
Ecole nationale Supérieure d'Informatique

Rapport du TP CRP

Thème :

Optimisation du jeu d'échecs en utilisant Iterative Deepening (IDS)

Réalisé par :
MAHMAHI Anis
BEKKAR ILHEM

Groupe :
SID

Promotion : 2019-2024

Introduction :

Minimax est un algorithme de prise de décision qui s'applique à la théorie des jeux, généralement utilisé dans les jeux à deux joueurs au tour par tour à somme nulle et à information complète. L'objectif de l'algorithme est de minimiser la perte maximum (c'est-à-dire dans le pire des cas).

Une version améliorée nommée élagage alpha/bêta existe et qui permet d'éviter de parcourir tous les nœuds sans affecter le résultat final. Cependant, cette amélioration n'est pas suffisante, en effet, dans certains cas, la portion élaguée reste négligeable devant la portion parcourue.

Dans le cadre de ce TP, il nous est demandé d'améliorer l'algorithme Minmax avec l'élagage alpha/bêta en qualité et en temps d'exécution.

Solution Proposée :

1. Amélioration du jeu en utilisant IDS:

L'algorithme de recherche en profondeur itérative (IDS) est un algorithme de recherche qui peut être utilisé pour améliorer l'algorithme MinMax, qui est un algorithme de prise de décision utilisé dans l'IA de jeu.
L'idée de base derrière IDS est de réaliser une recherche en profondeur de l'arbre de jeu, mais de limiter la profondeur de recherche à un certain niveau à chaque itération. La recherche débute à une profondeur de 0 et augmente progressivement la profondeur de bordure de 1 à chaque itération jusqu'à ce qu'une solution soit trouvée ou que le temps de recherche spécifié soit fini .

Lorsqu'il est utilisé conjointement avec MinMax, IDS peut améliorer les performances de l'algorithme en lui permettant de se concentrer sur les configurations les plus premières par les évaluer en premier, ensuite continuer la recherche plus profondément dans les branches les plus prometteuses de l'arbre de jeu.
Sans IDS, l'algorithme MinMax effectuera un parcours complet de l'arbre de jeu entier à chaque coup, ce qui peut être coûteux en termes de calcul pour les arbres volumineux. En limitant la profondeur de recherche à un certain niveau, IDS réduit le nombre de nœuds qui doivent être examinés, ce qui peut considérablement réduire le temps d'exécution de l'algorithme.

Code :

```
// profondeur initialisé à 1
  int hcc = 1;
// obtenir le temps actuelle
  int now = clock()/CLOCKS_PER_SEC;
// cc : c’est le temps ou la recherche doit s'arrêter
  int cc = now+duree;
  bool stop_time = false;
  while (!stop_time && n!=0){
        for (i=0;i<n;i++){
    cout = minmax_ab( &T[i], MIN, hcc, score, +INFINI, largeur,
estMax, nbp, cc );
          // mettre à jour les scores des configs pour les
ordonner.
              T[i].val = cout;
              if ( cout > score ) {
                        score = cout;
                            j = i;
                     }
              if ( cout == 100 ) {
                            printf("v"); fflush(stdout);
                            stop_time=true;
                            break;
                       }
              if (clock()/CLOCKS_PER_SEC>now+duree){
                     stop_time = true ; break;
                       }
                              }
              if(n==0) break;
 // ordonner les configs selon leurs scores obtenus au niveau
précédent
            qsort(T, n, sizeof(struct config), confcmp321);
        // incrémentation du profondeur
            hcc++;
      }
```

Test :

Config : fonction d'estim de B et N = 1 / le temps pour chaque coup = 3s / le nombre d'alternatives maximum = infiniti

	Version Initiale	IDS avec le temps
Profondeur moyen par coup	6	5.91
Le nombre de coups joués	177	58
le temps d'une partie	4 heurs et 17 min	250.501 s

Config : fonction d'estim 1 et 7 / le temps pour chaque coup = 3s / le nombre d'alternatives maximum = 5

	Version Initiale	IDS avec le temps
Profondeur moyen par coup	11	10.73
Le nombre de coups joués	49	160
le temps d'une partie	930.989 s	325.282 s

On peut remarquer que le temps a considérablement diminué et c'est grâce à la nation du temps pour effectuer un coup ce qui limite l'exploration de l'arbre, contrairement à la version initiale qui fait trop d'exploration pour finalement exploiter sa recherche. Donc, afin de maximiser la performance et minimiser le temps, il faut trouver un compromis entre l'exploration et l'exploitation pour effectuer des bons coups à temps réduit .

2. Utilisation des résultats intermédiaires :

Afin de minimiser le nombre de coups et le nombre de coupes Alpha/Beta, on a essayé d'exploiter les résultats intermédiaires du parcours **IDS.** Avant de passer d'un niveau à un autre, on essaie d'ordonner le tableau des configuration selon leur score obtenu au niveau précédent et donc elles soient ordonnés du plus prometteur au moins prometteur.

Test :

En testant le programme plusieurs fois, on remarque une amélioration claire de la qualité du jeu (**Nbr des coupes (alpha/Beta), La durée du jeu, Nombre des coups**).

Voici un exemple des résultats obtenus en prenant :

- Nombre d'alternatives maximum : 3
- Temps du coup : 3s
- Fonction d'estimation : B(7), N(1)

	Sans Sort	***Avec Sort***	***Facteur d'amélioration***
Nbr des coupes (alpha/Beta)	**25 238 933**	**1 065 618**	**23.68**
Temps d'exécution (s)	**1277.305**	**144.915**	**8.81**
Nbr des coups	**321**	**34**	**9.44**

3. Fonction d'estimation 7:

Le jeu d'échecs est un jeu stratégique basé sur plusieurs facteurs, et parmi ces derniers : le nombre du type de pièces possédées et la position de chaque type de pièce, d'où vient l'idée de créer une nouvelle fonction **estim7** qui prend en considération à la fois le **nombre** et la **position** de chaque type de pièce à chaque instant**.**

En effet, cette fonction n'est qu'une amélioration de la fonction **estim1** qui néglige la position de pièce qui représente un autre point de sa force.

Le principe c'est de donner des bonus pour les pièces qui sont bien placées et des pénalités (malus) pour les pièces mal placées. Le reste des pièces auront une valeur neutre de 0. Cette idée simple permet au moteur de faire des mouvements plus sensés.

Pour plus d'informations et l'idée derrière cette estimation voici l'article officiel de Tomasz Michniewski qui l'a créé.

https://www.chessprogramming.org/Simplified_Evaluation_Function

Exemple d'évaluation par position :

On encourage les pions à avancer par renforcer leurs poids de +50 s' ils sont en haut.	`[ 0, 0, 0, 0, 0, 0, 0, 0,`
	`50, 50, 50, 50, 50, 50, 50, 50,`
	`10, 10, 20, 30, 30, 20, 10, 10,`
	`5, 5, 10, 25, 25, 10, 5, 5,`
	`0, 0, 0, 20, 20, 0, 0, 0,`
	`5, -5,-10, 0, 0,-10, -5, 5,`
	`5, 10, 10,-20,-20, 10, 10, 5,`
	`0, 0, 0, 0, 0, 0, 0, 0 ]`
	La table des poids associée aux pions.

On évite les coins et les frontières, et on préfère les carrés b3, c4, b5, d3.	`[ -20,-10,-10,-10,-10,-10,-10,-20,`
	`-10, 0, 0, 0, 0, 0, 0,-10,`
	`-10, 0, 5, 10, 10, 5, 0,-10,`
	`-10, 5, 5, 10, 10, 5, 5,-10,`
	`-10, 0, 10, 10, 10, 10, 0,-10,`
	`-10, 10, 10, 10, 10, 10, 10,-10,`
	`-10, 5, 0, 0, 0, 0, 5,-10,`
	`-20,-10,-10,-10,-10,-10,-10,-20`
	`]`
	La table des poids associée aux tours.

Les chevaliers, on les encourage simplement à aller au centre. Rester aux frontières est une mauvaise idée.	`[ -50,-40,-30,-30,-30,-30,-40,-50,`
	`-40,-20, 0, 0, 0, 0,-20,-40,`
	`-30, 0, 10, 15, 15, 10, 0,-30,`
	`-30, 5, 15, 20, 20, 15, 5,-30,`
	`-30, 0, 15, 20, 20, 15, 0,-30,`
	`-30, 5, 10, 15, 15, 10, 5,-30,`
	`-40,-20, 0, 5, 5, 0,-20,-40,`
	`-50,-40,-30,-30,-30,-30,-40,-50,]`
	La table des poids associée aux chevaliers.

Evaluation par le nombre et le poids des pièces :

Les poids sont fixés comme suit:

- ➔ pion:100
- ➔ cavalier:320
- ➔ fou: 330
- ➔ tour:500
- ➔ la reine:900
- ➔ le roi : 20000

Code :

```
int estim7( struct config *conf ){
      int i, j=0; float ScrQte=0;
int pionB = 0, pionN = 0, cB = 0, cN = 0, fB = 0, fN = 0, tB = 0, tN =
0, nB = 0, nN = 0,rB=0,rN=0;
      // parties : nombre de pièces et occupation du centre
      for (i=0; i<8; i++)
         for (j=0; j<8; j++) {
switch (conf->mat[i][j]) {
 case 'p' : pionB++;
                ScrQte += piece_square_table[0][i*8+j];
break;
 case 'c' : cB++;
                ScrQte += piece_square_table[1][i*8+j];
break;
case 'f' : fB++;
                ScrQte += piece_square_table[2][i*8+j];
break;
 case 't' : tB++;
                ScrQte += piece_square_table[3][i*8+j];
break;
case 'n' : nB++;
                ScrQte += piece_square_table[4][i*8+j];
break;
case 'r' : rB++;
                ScrQte += piece_square_table[5][i*8+j];
                if (nN+nB==0){    // end game
ScrQte += piece_square_table[6][conf->xrB*8+conf->yrB];   }
         else{ // mid - early king
       ScrQte += piece_square_table[5][conf->xrB*8+conf->yrB];   }
                break;
case -'p' : pionN++;
                ScrQte -= mirror[0][i*8+j];              break;
 case -'c' : cN++;
                ScrQte -= mirror[1][i*8+j];              break;
case -'f' : fN++;
                ScrQte -= mirror[2][i*8+j];              break;
case -'t' : tN++;
                ScrQte -= mirror[3][i*8+j];              break;
case -'n' : nN++;
                ScrQte -= mirror[4][i*8+j];              break;
case -'r' : rN++;
               ScrQte -= mirror[5][i*8+j];
               if (nN+nB==0){// end game king
            ScrQte -= mirror[6][conf->xrN*8+conf->yrN];
         }else{ // mid - early king
    ScrQte -= mirror[5][conf->xrN*8+conf->yrN]; }
                break;}
```

```
 }
      // Somme pondérée de pièces de chaque joueur.
ScrQte += ( (pionB*100 + cB*320 + fB*330 + tB*500 + nB*900+ 20000*rB) -
(pionN*100 + cN*320 + fN*330 + tN*500 + nN*900+ 20000*rN) );
      // Le facteur 1/100 pour ne pas sortir de l'intervalle ]-100 ,
+100[
      return ScrQte/100;
} // fin de estim7
```

Test :

Dans ce qui suit nous allons recenser les résultats (les gagnants) des 5 différentes et successives parties d'échecs oú nous avons mis la fonction estim7 (joueur Noir) contre d'autres fonctions d'estimations (joueur Blanc) pour tester l'efficacité de notre solution.

Les tests suivants ont été fait en spécifiant le temps a chaque joueur égale à 1s et le nombre d'alternatives maximum à considérer dans chaque coup est 5 pour minimiser un peu le temps d'exécution.

Estim / Partie	1	2	3	4	5
1 vs 7	7	7	1	7	1
3 vs 7	1	1	1	1	1
4 vs 7	7	7	7	1	1
5 vs 7	7	7	7	7	7

On remarque que la fonction d'estimation 7 donne toujours de meilleurs résultats que la 5eme fonction. Contrairement a la 3eme est toujours mieux que la 7eme.

La comparaison avec la première et la quatrième fonction, nous montre que la 7eme fonction a permit le joueur N de gagner ⅗ des matches joués.

4. Aucun coup possible :

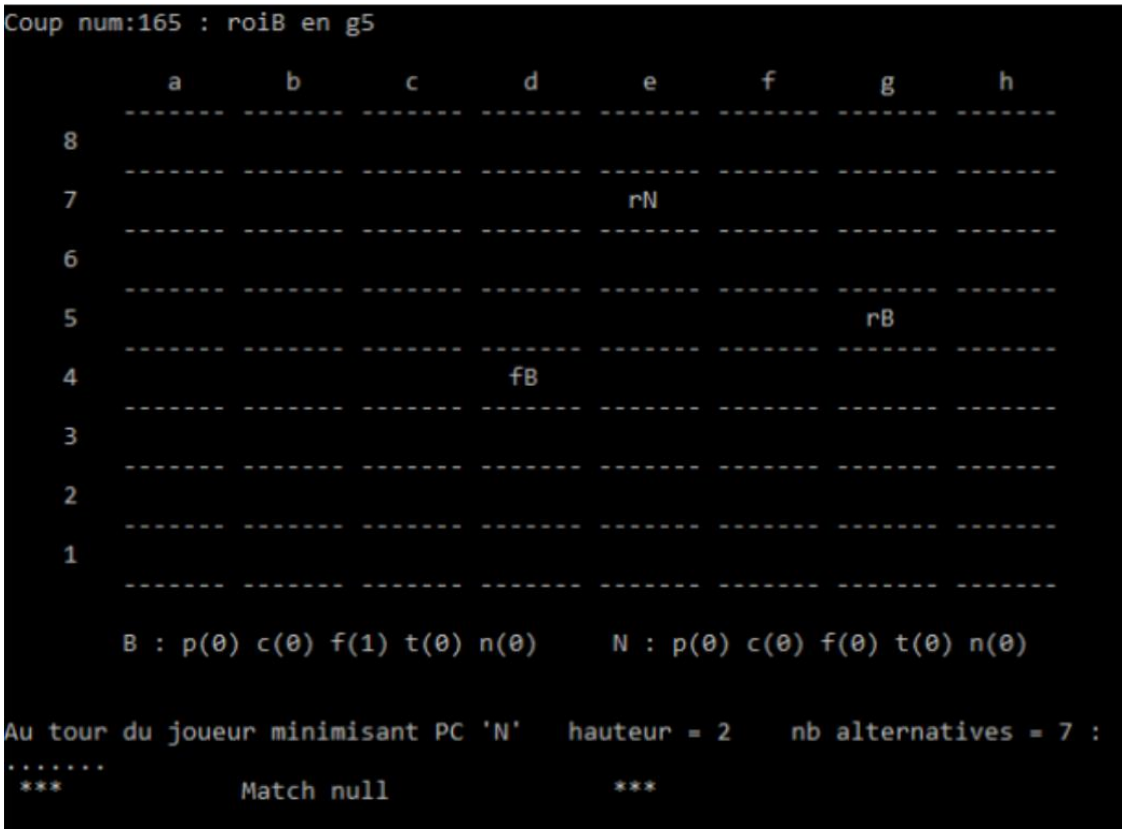
Afin de clôturer une partie dans le cas d'un nul, on a complété l'implémentation de la fonction *AucunCoupPossible* (code dans jeu.h).

Cette fonction vérifie s'il s'agit bien d'un match nul ou pas :

- Si le nombre de successeurs possibles d'une configuration est nul (ce qui veut dire que le joueur n'a aucun coup à jouer) .
- Sinon on évalue le matériel restant, si on ne dispose pas assez de pièces pour faire un échec et mat (roi vs roi , roi +(fou ou cavalier) vs roi).
- Sinon, on vérifie si la configuration actuelle a été générée plusieurs fois.

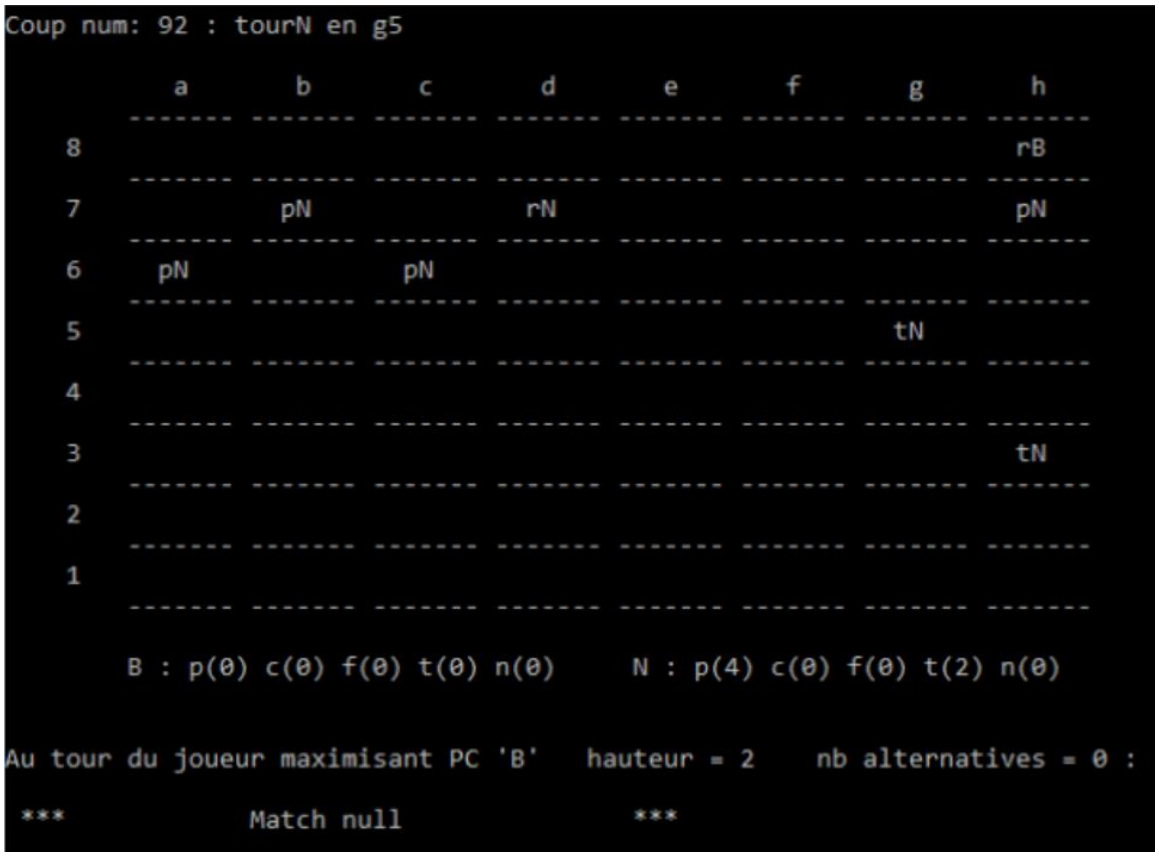
Test :

a) Match nul à cause du matériel insuffisant :



La partie s'arrête car il ne reste qu'un roi noir et un fou et roi blanc, pas de possibilité de se départager dans ce cas-là, donc la partie se termine avec égalité pour matériel insuffisant.

b) Match nul à cause d'inexistence de successeurs :



Les blancs n'ont rien d'autre que leur roi, et celui-ci n'a pas de positions ou se déplacer (les cases voisines sont menacées par les pièces noires), de plus le roi blanc n'est pas en échec, donc fin de partie avec nul.

5. L'effet Horizon :

En général, une recherche de jeu qui s'arrête à une profondeur fixe pose un problème. Dans une recherche d'échecs, par exemple, le dernier coup des blancs peut-être la dame prend le cavalier, et la fonction d'évaluation rapportera que les blancs sont en haut d'un cavalier. Si la recherche examinait un mouvement plus profond, elle verrait que les noirs ont la réponse le pion prend la reine et se rend compte que les noirs sont en train de gagner.

Solution proposée :

La recherche de quiescence est une recherche supplémentaire, commençant aux nœuds frontières instables de la recherche principale, qui tente de résoudre le problème de l'effet horizon.

C'est une extension de la fonction d'évaluation pour différer l'évaluation jusqu'à ce que la position soit suffisamment stable pour être évaluée et elle tente de chercher les positions « instables » à une plus grande profondeur que les

positions « stables » pour s'assurer qu'il n'y a pas de pièges cachés et pour obtenir une meilleure estimation de sa valeur.

Conclusion :

À travers ce rapport, nous avons présenté tout le travail réalisé dans ce TP et quelques idées d'amélioration non encore réalisées dans le cadre du module CRP (Complexité & Résolution de Problèmes). À la fin de chaque amélioration, nous avons présenté différents tests pour illustrer l’effet de l’amélioration implémentée et montrer l’efficacité de notre solution.