Faure Rémi
Mouret Clément
Nguyen Anh-Tai
EFREI I2 A2

Théorie des Jeux et jeu d'échecs

Sommaire

1 Introduction

Ce document présente le bilan de nos recherches sur la théorie des jeux appliquées au jeu d'échecs ainsi que le résultat obtenu après la conception d'un jeu d'échecs sur ordinateur.

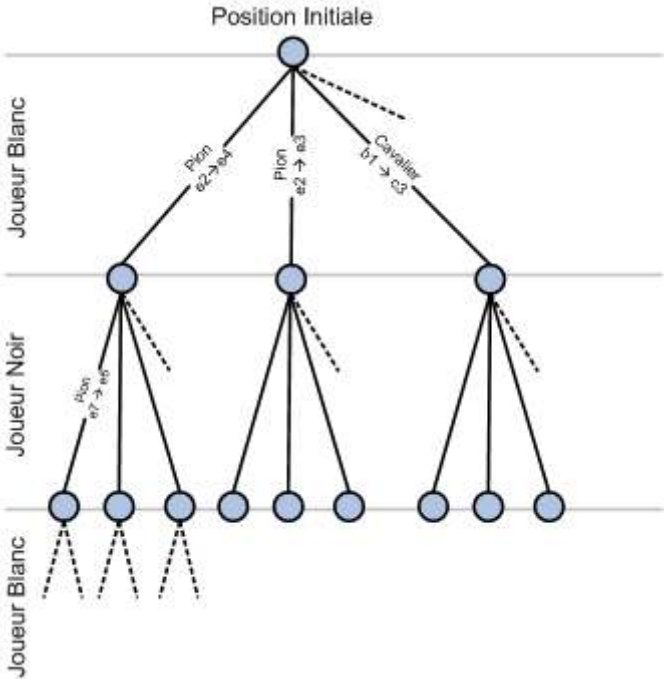
Notre analyse théorique est construite autour des deux aspects principaux suivants : la théorie des jeux et son application à la programmation d'un jeu d'échecs. Les principaux algorithmes de la théorie des jeux (MiniMax, Alpha-Beta, SSS*, MTD(f) …) seront présentés et analysés afin de comprendre leur fonctionnement et de comparer leur efficacité. Nous évoquerons aussi les optimisations possibles permettant d'améliorer les performances de l'intelligence artificielle. Ces optimisations concernent à la fois le temps de réflexion et la qualité des coups joués par l'ordinateur. Nous aborderons ensuite dans une deuxième partie, les aspects plus propres aux jeux d'échecs comme la représentation d'un échiquier en machine, les ouvertures et fermetures, et la fonction d'évaluation.

La troisième partie de ce rapport est consacrée à la phase de conception de notre jeu d'échecs baptisé diamondChess. Son but est de présenter les solutions choisies, les problèmes rencontrés lors de la phase de développement et enfin la version finale du logiciel.

2 Théorie des jeux

2.1 Représentation du déroulement d'une partie

Le jeu d'échecs est un jeu où les deux joueurs jouent séquentiellement. Le but des deux joueurs est opposé et incompatible. Le joueur blanc veut mettre le roi noir mat tandis que le joueur noir veut mettre le roi blanc mat. Tour à tour, un joueur puis l'autre choisit parmi tous les coups possibles celui qui lui paraît le meilleur pour parvenir à son objectif. Ainsi il est possible de représenter une partie de jeu par un arbre. Chaque sommet représente une position possible et les arcs sont tour à tour les coups jouables par le joueur blanc puis par le joueur noir puis le blanc…



Il est donc possible de programmer une Intelligence Artificielle du jeu d'échecs qui va réfléchir à la conséquence de ses actes en regardant les mouvements possibles d'un joueur pour pouvoir jouer à sa place.

2.2 Algorithme de recherche

Nous avons vu que le déroulement d'une partie d'échecs peut être représenté par un arbre. Il faut partir du principe que les deux adversaires sont là pour jouer et pour gagner. Donc à chaque coup, ils vont jouer leur meilleur coup possible afin de mettre toutes les chances de leur côté, et tenter de gagner la partie.

Notre IA doit donc tenter d'analyser ce que son adversaire va faire aux coups suivants afin de déterminer le meilleur coup à jouer. Plusieurs algorithmes existent et nous allons essayer de voir comment ils fonctionnent.

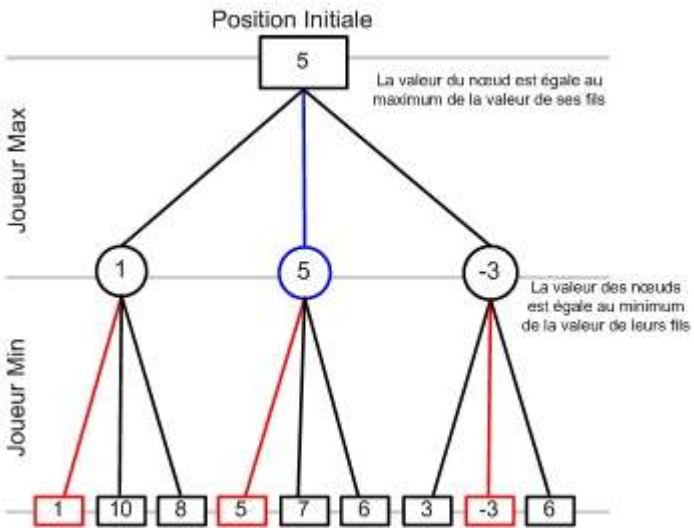
2.2.1 L'algorithme MiniMax

On rappelle qu'une fonction d'évaluation prend en entrée une position dans un jeu et donne en sortie une évaluation numérique pour cette position. L'évaluation est d'autant plus élevée que la position est meilleure. Si une fonction d'évaluation est parfaite, il est inutile d'essayer de prévoir plusieurs coups à l'avance. Toutefois, pour les jeux complexes comme les Echecs, on ne connaît pas de fonction d'évaluation parfaite. On améliore donc un programme si à partir d'une bonne fonction d'évaluation on lui fait prévoir les conséquences de ses coups plusieurs coups à l'avance (voir le paragraphe 3.2 consacré à la fonction d'évaluation).

L'hypothèse fondamentale du MiniMax est que l'adversaire utilise la même fonction d'évaluation que le programme pour déterminer si une position lui est favorable ou non.

Notons Max le joueur que l'on cherche à faire gagner (la machine) et Min son adversaire (l'humain). Le joueur Min joue logiquement et ne va pas perdre une occasion de gagner. Si pour gagner le joueur Max tente de maximiser son score, le joueur Min va tenter de maximiser le sien (c'est à dire de minimiser le score de Max).

Voyons comment fonctionne l'algorithme. Le joueur Max commence à jouer.



2.2.1.1 Algorithme

```
MiniMax
Entrées : position P, entier profondeur
Sortie  : entier valeurDeLaPosition
{
  Si P est une feuille (i.e. profondeur = profondeur maximale)
  {
    valeurDeLaPosition = eval(P)
    sortir
  }

  Pour tous les fils F de P faire
  {
    valeurDuFilsCourant = MiniMax(F, profondeur+1)
    si P nœud Max alors
    {
      Si valeurDuFilsCourant > valeurActuelleDuNoeudP
      ou 1er des fils examinés alors
      {
        ValeurActuelleDuNoeudP = valeurDuFilsCourant
      }
    }
    sinon (c'est un nœud min)
    {
      Si valeurDuFilsCourant < valeurActuelleDuNoeudP
      ou 1er des fils examinés alors
      {
        ValeurActuelleDuNoeudP = valeurDuFilsCourant
      }
    }
  }

  valeurDeLaPosition = valeurActuelleDuNoeudP
  sortir
}
```

2.2.1.2 Inconvénients de l'algorithme

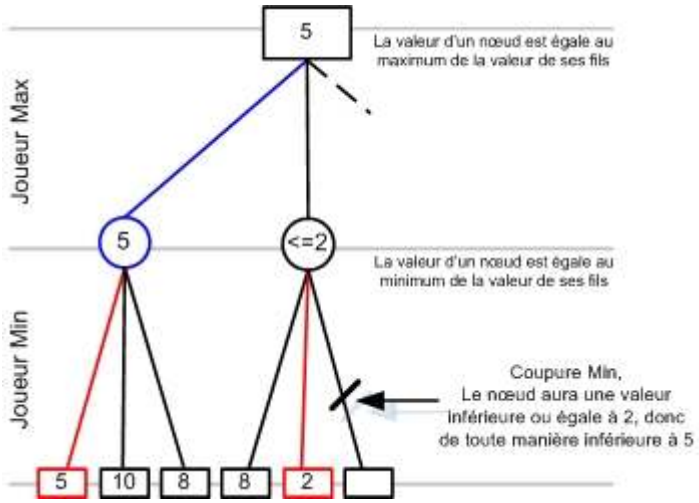
L'inconvénient majeur de cet algorithme est qu'il explore toutes les branches de l'arbre sans distinction alors que certaines branches n'apporteront rien.

Le facteur de branchement d'un jeu comme les échecs est de l'ordre de 35. Si on calcule tous les coups possibles avec une profondeur de 100, on obtient un arbre d'environ 35^{100} soit plus d'éléments que d'atomes dans l'Univers (de l'ordre de 10^{100}). Il est donc évident qu'il est impossible d'explorer tous les coups possibles à une si grande profondeur.

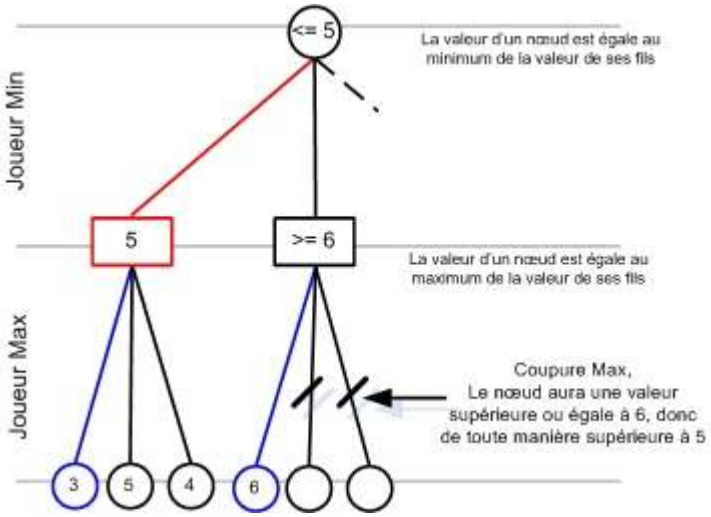
2.2.2 L'élagage Alpha-Beta

Une amélioration du MiniMax est de faire un élagage de certaines branches de l'arbre. En effet, il y a certaines situations dans lesquelles il n'est pas nécessaire d'examiner toutes les branches de l'arbre. Si la valeur d'un nœud atteint un seuil, il est inutile de continuer à explorer les descendants de ce nœud car leurs valeurs n'interviendront pas. Voyons cela sur un exemple :

Coupure Min :



Coupure Max :



2.2.2.1 Algorithme

```
Alpha_Beta
Entrées : position P, entier profondeur, entier alpha, entier beta
Sortie  : entier valeurDeLaPosition
{
  Si P est une feuille (i.e. profondeur = profondeur maximale)
  {
    valeurDeLaPosition = eval(P)
    sortir
  }
  sinon
  {
    si P est un nœud max alors
    {
      Pour tous les fils F de P et tant que alpha < beta faire
      {
        valeurDuFilsCourant = Alpha_Beta(F, profondeur+1,alpha,beta)
        alpha = max(alpha, valeurDuFilsCourant)
      }
      valeurDeLaPosition = alpha
      sortir
    }
```

```
        sinon
        {
          Pour tous les fils F de P et tant que alpha < beta faire
          {
            valeurDuFilsCourant = Alpha_Beta(F, profondeur+1,alpha,beta)
            beta = min(beta, valeurDuFilsCourant)
          }
          valeurDeLaPosition = beta
          sortir
        }
      }
    }
```

2.2.2.2 Avantages par rapport au MiniMax

L'avantage de cet algorithme est qu'il donne exactement les mêmes résultats que l'algorithme MiniMax tout en étant beaucoup plus rapide. Si les nœuds sont correctement ordonnés la vitesse de l'algorithme est accrue du fait que les coupes surviennent plus tôt. Ainsi pour un même temps d'exécution, l'algorithme Alpha-Beta permettra d'aller à une profondeur plus grande.

2.2.3 Algorithme Nega-Alpha

NegaAlpha est une optimisation simple de l'algorithme Alpha-Beta. En effet, il ne sert à rien de savoir si on est sur un nœud min ou un nœud max, pour savoir s'il faut minimiser ou maximiser l'évaluation, il suffit simplement d'inverser les signes des évaluations à chaque niveau, et ainsi on cherche toujours à maximiser.

2.2.3.1 Algorithme

```
Alpha_Beta
Entrées : position P, entier profondeur, entier alpha, entier beta
Sortie  : entier valeurDeLaPosition
{
  Si P est une feuille (i.e. profondeur = profondeur maximale)
  {
    valeurDeLaPosition = eval(P)
    sortir
  }
  sinon
  {
    Pour tous les fils F de P et tant que alpha < beta faire
    {
      valeurDuFilsCourant = - Alpha_Beta(F, profondeur+1,-beta,-alpha)
      alpha = max(alpha, valeurDuFilsCourant)
    }
    valeurDeLaPosition = alpha
    sortir
  }
}
```

2.2.4 Algorithme Alpha-Beta avec mémoire.

Les algorithmes tels MTD(f) nécessitent de relancer alpha-beta avec différente valeur d'alpha et de bêta, il est donc nécessaire de trouver un moyen de gagner du temps et d'éviter de réévaluer les nœuds suivants, ce qui est coûteux en temps. Un moyen simple pour éviter cette réévaluation systématique est de mettre en place une table de transposition. Ainsi l'algorithme deviendrait :

2.2.4.1 Algorithme

```
Alpha_Beta_Avec_Memoire
Entrées : position P, entier depth, entier alpha, entier beta
Sortie  : entier valeurDeLaPosition
{
  if (retrieve(Position) == true) {

    if (position.lowerbound >= bêta)
      return position.lowerbound;
    if (position.upperbound <= alpha)
      return position.upperbound;
    alpha = max(alpha, position.lowerbound);
    bêta = min(bêta, position.upperbound);
  }

  if (game over or depth <= 0)
    return winning score or eval();

  int a = alpha ;

  move bestmove ;
  int current = -INFINITY;
  for (each possible move m) {
    make move m;
    int score = - Alpha_Beta_Avec_Memoire (depth - 1, -bêta, -a)
    unmake move m;

    if (score >= current) {
      current = score;
      bestmove = m;
      if (score >= a){
        a = score;
        if (score >= bêta)
          break;
      }
    }
  }

  if (g <= alpha) {
    position.upperbound = current;
    store position.upperbound;
  }
  if (g > alpha && g < bêta){ //impossible dans le cas du mtd(f)
    position.lowerbound = current;
    position.upperbound = current;
    store position.lowerbound, position.upperbound;
  }
  if (g >= bêta) {
    position.lowerbound = current;
    store position.lowerbound;
  }

  return current;
}
```

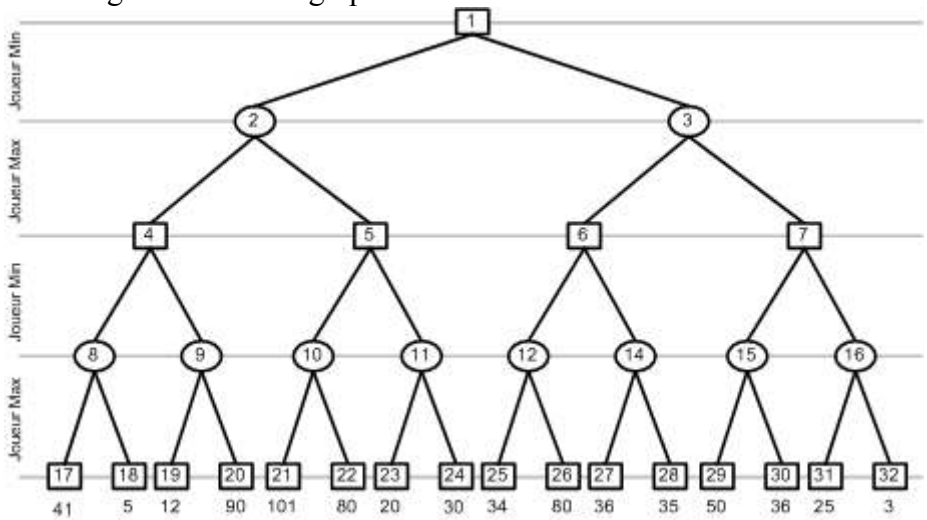
2.2.5 Autres algorithmes

2.2.5.1 SSS*

SSS* est un algorithme de recherche dans les arbres de jeu de type "meilleur d'abord". Chaque nœud de l'arbre représente un groupe d'arbres solutions (stratégie). Cet algorithme raffine le groupe ayant la limite supérieure la plus élevée par rapport aux valeurs des feuilles qu'il contient. L'algorithme SSS* est nettement moins connu et moins utilisé que l'algorithme α-β bien qu'il a été démontré qu'il lui est théoriquement supérieur. Mais SSS* nécessite une quantité importante de mémoire.

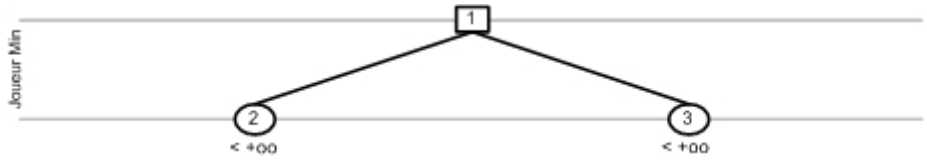
On rappelle qu'une stratégie indique au joueur Max ce qu'il doit jouer dans tous les cas. S'il s'agit d'un nœud Max, il a toute liberté de choix et jouera donc le coup indiqué par la stratégie. Si c'est un nœud Min, la stratégie contient tous ses fils et envisage donc toutes les réponses éventuelles de l'adversaire. Si Max respecte une stratégie, il est assuré d'aboutir à une des feuilles de la stratégie. Si l'on se place dans le cas habituel où l'arbre de jeu est développé jusqu'à une profondeur n. Le mérite des positions terminales est estimé par une fonction d'évaluation. La valeur d'une stratégie pour Max est donc le minimum des valeurs des feuilles de cette stratégie, gain assuré contre toute défense de Min. Le but de SSS* est d'exhiber la stratégie de valeur maximum pour Max. Sa valeur sera, par définition, la valeur minimax de l'arbre.

Appliquons l'algorithme sur le graphe suivant :

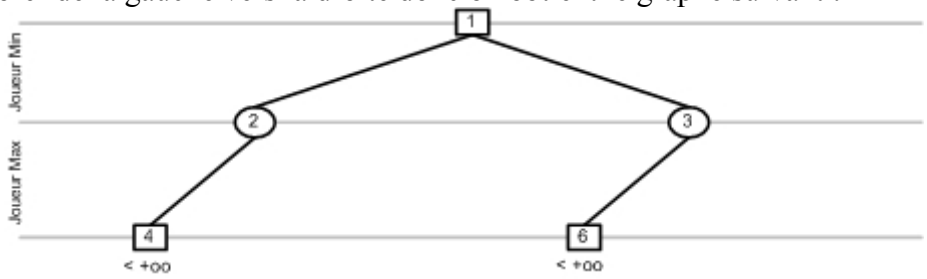


Les nœuds sont numérotés de 1 à 32. Seules les évaluations des feuilles de l'arbre sont calculées directement par la fonction d 'évaluation. L'algorithme va nous permettre de trouver l'évaluation du nœud 1. On se limitera dans cet exemple à un arbre binaire mais le principe reste le même sur un arbre n-aire.

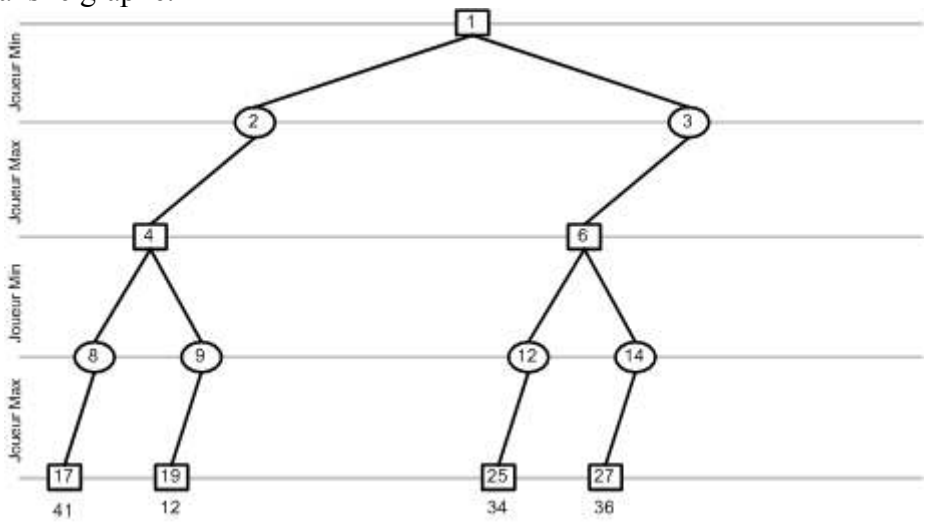
Etape 1 : Au départ, l'arbre n'est pas connu. C'est un nœud de type Min donc l'algorithme nous dit qu'il faut développer tous ses fils. On génère donc les deux fils du nœud 1. Les nœuds 2 et 3 ne sont pas encore évalués et ont donc une évaluation inférieure à l'infini.



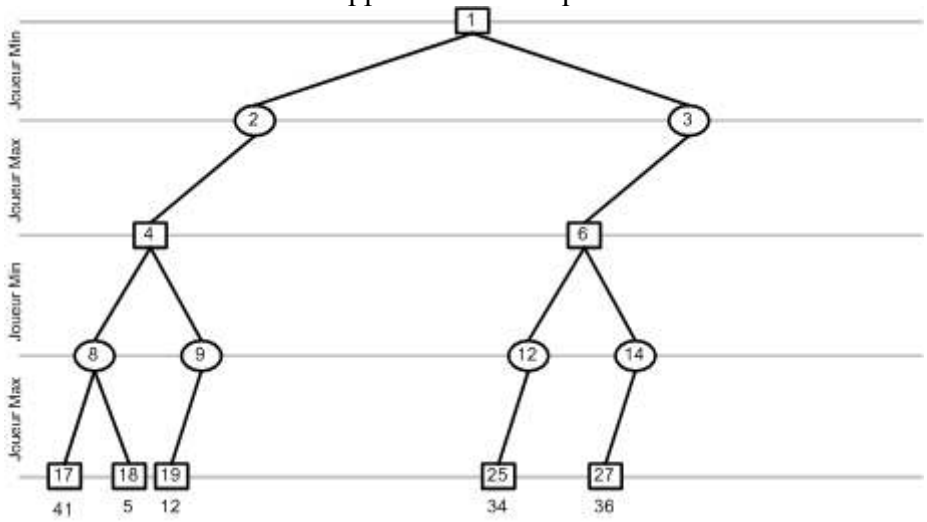
Etape 2 : l'algorithme nous dit ensuite de n'évaluer qu'un seul fils des nœuds de type Max. Et de les explorer de la gauche vers la droite donc on obtient le graphe suivant :



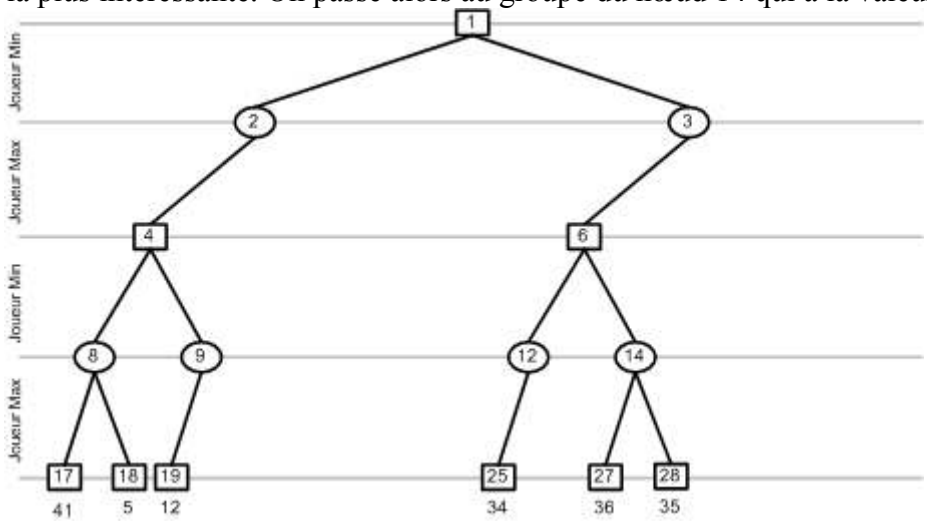
Etape 3 : On recommence les étapes une et deux jusqu'à ce qu'il n'y ait plus de valeurs + l'infini dans le graphe.



Etape 4 : On évalue alors plus en détail le groupe qui à la valeur la plus élevée, donc ici le nœud 17 qui a la valeur 41. On développe le nœud 18 qui a la valeur 5.



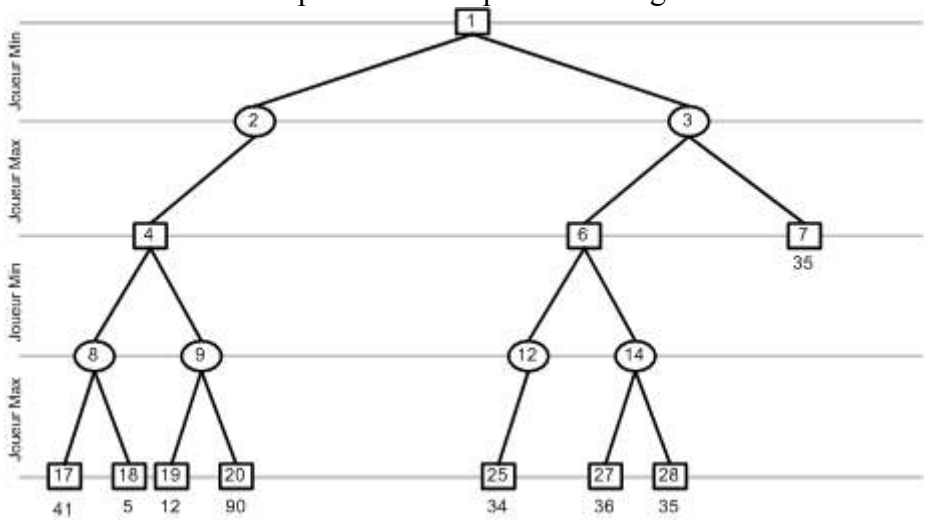
Etape 5 : La moyenne du groupe du nœud 8 passe à (41 + 5) / 2 = 23. Ce groupe n'est plus la stratégie la plus intéressante. On passe alors au groupe du nœud 14 qui a la valeur 36.



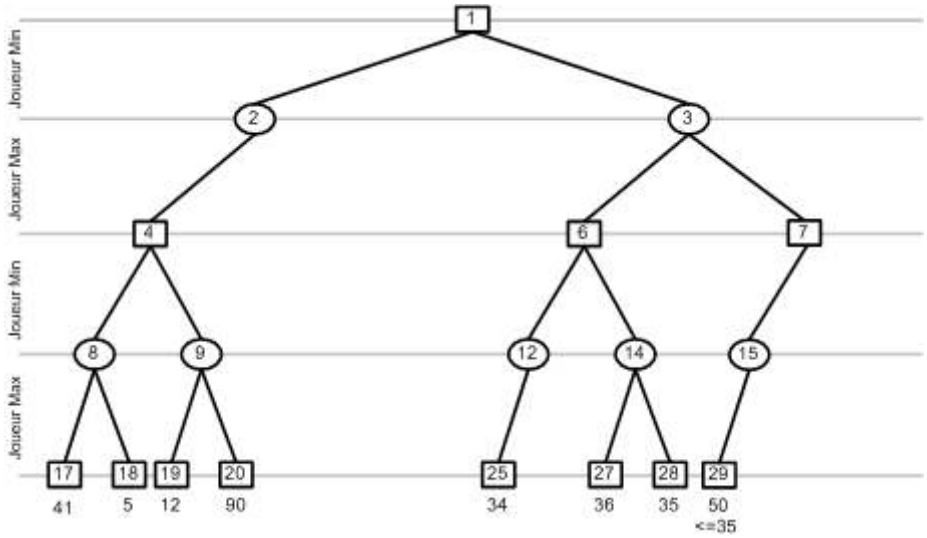
Le joueur Max prend le minimum des valeurs des stratégies des nœuds inférieurs. Donc n14 aura la valeur 35, et le nœud 12 une valeur inférieure ou égale à 34. Min prend le maximum des valeurs des stratégies, et donc n6 aura pour valeur 35, et est alors marqué comme résolu puisque qu'aucune valeur des nœuds fils de 6 ne pourra augmenter sa valeur.

Etape 6 : Le nœud 3 a donc pour limite supérieure 35. C'est encore le groupe le plus avantageux donc on développe le fils droit de 3, en propageant la limite. On explore le nœud 7 est espérant vérifier que la valeur du nœud 3 est toujours supérieure aux limites.

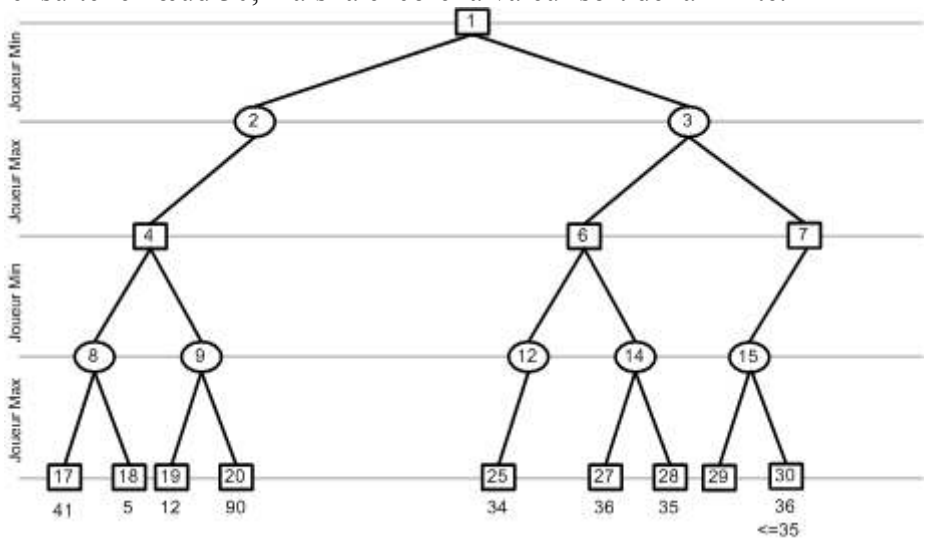
On procède de même que pour l'exploration du nœud racine avec cette fois la limite 35 et non plus + l'infini. On peut terminer la vérification dès que nous concluons que le nœud 3 est la racine d'un arbre dont la valeur supérieure est supérieure ou égale à 35.



Etape 7 : on explore les fils du nœud 7. Avec la limite supérieure de 35. n20 a une valeur supérieure à la limite.



On explore ensuite le nœud 30, mais là encore la valeur sort de la limite.



Donc n15 a pour valeur 36. Le nœud 7 est considéré comme résolu. Un des fils a été complètement développé et les autres sous-arbres ont des valeurs inférieures, on peut donc conclure que la valeur de la racine n3 est 35.

On peut donc en conclure que le meilleur déplacement est vers le nœud 3.

2.2.5.1.1 Algorithme de SSS :*

```
Soit n0 le nœud à évaluer
G = ((n0,vivant,+oo)) ;
Tant que Premier(G) n'est pas de la forme (n0,résolu,v)) faire
  (n,t,e) = ExtraitPremier(G) ;
  si t = vivant alors
    suivant le type de n faire
      cas où n est terminal : Insérer((n,résolu,min(e,h(n))),G);
      cas où n est Max :
        soit (f1, … ,fn) les fils de n ;
        pour i allant de 1 à K faire Inserer((fi(n),vivant,e),G);
      cas où n est Min : Inserer((f1(n),vivant,e),G) ;
  sinon
    si n est de type Min alors
      Inserer((p(n),resolu,e),G);
      Supprimer de G tous les etats dont le nœud est un succeseurs de p(n)
    Sinon
      Si n a un frère suivant alors Inserer((frere(n),vivant,e),G);
      Sinon Inserer((p-n),resolu,e),G);
Retourner v ;
```

2.2.5.2 Scout

L'algorithme Scout a été élaboré par J. Pearl en 1990. Dans la pratique, son efficacité est en général inférieure à un algorithme α-β. Mais dans certains cas, il peut être plus performant. L'idée principale est de disposer d'un moyen heuristique de comparer une valeur avec la valeur d'un nœud sans obligatoirement l'évaluer. Ainsi de nombreuses branches de l'arbre peuvent être éliminées de la recherche.

2.2.5.3 MTD(f)

L'algorithme MTD(f) est une évolution des algorithmes du type MiniMax et Alpha-Beta. Son but est d'augmenter le nombre d'élagage et ainsi d'augmenter la vitesse de l'évaluation de l'arbre de jeux.

Il a été démontré que dans 90 % des positions, il est rare que la valeur d'une position évolue beaucoup. Il est donc possible d'initialiser les valeurs α et β à la valeur de la position courante ± ε si on considère que une position ne peut pas gagner ± ε lors de son évaluation. MTD(f) est basé sur le principe d'appel à un algorithme du type α-β à partir d'un intervalle réduit. Dans le cas ou ε = 1, on parle de fenêtre nulle.

Tous les coups qui seront en dehors de la fenêtre seront immédiatement élagués. La recherche α-β à partir d'une fenêtre réduite est donc très rapide car peu de branches sont explorées. L'utilisation d'une fenêtre définissant les bornes α et β est juste uniquement si l'hypothèse initiale est respectée, c'est-à-dire qu'on ne peux pas perde ou gagner plus de ε. Dans le cas contraire, il faut relancer l'algorithme α-β à partir d'une fenêtre plus large.
Cette technique peut paraître coûteuse en temps d'évaluation d'un nœud. En effet un nœud est susceptible d'être évalué plusieurs fois. Mais si on utilise des tables de transposition (voir 2.4) permettant de ne pas réévaluer un nœud déjà évalué lors d'une recherche précédente, certains nœud ne sont pas examinés à nouveau, économisant ainsi du temps de calcul.

On parle d'algorithme multipasses.

Voici l'algorithme MTD(f) :

```
MTD(racine,f,)
Entrées : racine de l'arbre, borne de la fenêtre d'étude
Sorties : valeur de la position
{
       g = f
       BorneHaute = +∞
BorneBase = +∞

Tant que BorneHaute > BorneBasse faire
{
              si g = BorneBasse alors
β = g + 1
              sinon
                     B = g
              g = Alpha_Beta_Avec_Memoire(racine, β – 1, β)
              si g < β alors
                     BorneHaute = g
              sinon
BorneBasse = g
}
retourner g
}
```

Cet algorithme appelle autant de fois que nécessaire l'algorithme α-β afin de construire un encadrement de la valeur de la position courante. Quand la borne basse et la borne haute se rencontrent, c'est-à-dire que la borne haute est inférieure ou égale à la borne basse, l'algorithme s'arrête et l'évaluation est terminée.

Cette méthode est particulièrement efficace quand le paramètre initial f de l'algorithme est proche de la valeur réelle de la position. On utilise généralement comme valeur de f une valeur retournée par l'algorithme lors d'une itération précédente.

2.3 Optimisations possibles

2.3.1 L'approfondissement itératif

Construire l'arbre avec le principe de l'approfondissement consiste à en créer tous les nœuds à chaque niveau de profondeur jusqu'à trouver la solution finale au problème. On calcule ainsi tous les nœuds de profondeur n avant de passer à ceux de profondeur n+1.

Etant donné que le nombre de coups possible est proportionnel au niveau de profondeur de l'arbre, le nombre de nœuds explorés à une profondeur n sera négligeable face au nombre d'itération n+1.

2.3.2 Le coup qui tue

L'ordre dans lequel on considère les coups a une grande influence sur les performances de l'algorithme. Une des heuristiques les plus utilisées est de considérer que si un coup est très supérieur aux autres dans une branche, il peut être intéressant de l'explorer en premier dans les autres branches. Ce coup est appelé "coup qui tue". La plupart des jeux d'échecs, utilisent cette heuristique en mémorisant le meilleur coup à chaque profondeur. On peut généraliser cette heuristique en gardant tous les mouvements légaux rencontrés dans l'arbre de recherche et en leur accordant une note. Cette note est proportionnelle à la profondeur du sous arbre exploré. La note du meilleur coup est augmentée de 2ProfondeurMax−p où p est la profondeur à laquelle le noeud a été exploré. Ensuite, on explore le noeud suivant l'ordre de cette note.

2.3.3 La recherche aspirante

Lorsqu'on utilise l'algorithme alpha-beta, au lieu de démarrer avec les valeurs de +oo et –oo à chaque parcours de niveau de profondeur, on initialise la recherche avec des valeurs issues de recherches précédentes. On réduit ainsi le champ de recherche pour gagner en vitesse d'exécution. La valeur retournée sera correcte si elle est comprise entre les valeurs d'initialisation.

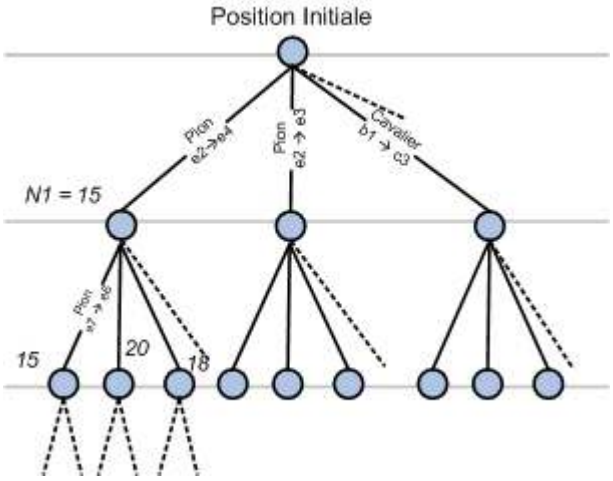
2.3.4 Alpha-Beta trié

Cela consiste à donner une « pré-note » à chaque nœud parcouru. Le parcours devra alors se faire de manière à avoir le plus de coupures possibles en utilisant une méthode de parcours de type "meilleur d'abord". En choisissant ainsi quels nœuds on va parcourir, nous pouvons ainsi augmenter le niveau de profondeur parcouru sans avoir à augmenter le nombre de feuilles parcourues.

2.4 Tables de transposition

L'évaluation d'un nœud de l'arbre de jeu repose sur une fonction d'évaluation complexe et donc coûteuse en temps de calcul. Les tables de transposition conservent la valeur de chacune des positions rencontrées au cours de l'évaluation d'un nœud afin de pouvoir les réutiliser lors de chaque évaluation successive de l'arbre. Le temps de calcul est ainsi largement amélioré. Les informations stockées pour une position sont la position, la valeur calculée par la fonction d'évaluation, le meilleur coup à jouer pour cette position et la profondeur à laquelle le nœud a été évalué.

Exemple :



Soit le N1 dont les informations stockées par l'ordinateur sont :

Position	N1
Evaluation	15
Meilleur coup	Gauche
Profondeur	2

Imaginons que lors d'une recherche future l'ordinateur rencontre de nouveau la position N1. Si la distance entre la racine et le nœud de l'arbre (profondeur) est plus grande que la profondeur stockée pour cette position dans la table de transposition, l'ordinateur n'évaluera pas ce nœud et utilisera directement la valeur calculée précédemment. Dans le cas contraire, l'ordinateur réévaluera la position. En effet la valeur de cette position peut être affinée par l'algorithme de recherche de type MiniMax vu que la distance entre le nœud et les feuilles de l'arbre est plus grande. L'ordinateur est donc capable de voir plus de coups en avance et de calculer un meilleur coup à jouer.

Pour stocker les tables de transposition, il est conseillé d'utiliser des tables de hachage permettant une recherche rapide. Les positions de l'échiquier sont ainsi indexées et accessibles rapidement en mémoire par l'ordinateur en évitant une recherche itérative longue et coûteuse en temps de calcul. On utilise donc un nombre (clef de hachage) caractérisant cette position.

Le problème des tables de hachage est qu'il impossible d'utiliser un index de trop grande taille. En effet une telle structure utiliserait beaucoup trop de mémoire dans le cas du jeu d'échecs. Il faut donc diminuer le nombre de positions stockées en mémoire. Généralement on utilise comme index de position la clef de hachage tronquée. C'est-à-dire que si on utilise des clefs de hachage sur 32 bits (un clef = une position unique), on utilise par exemple que les 16 derniers bits de ce nombres. En diminuant le nombre de position stockée, une case mémoire de la structure ne correspond plus uniquement à une position de l'échiquier. Il est possible de résoudre ce conflit en comparant la valeur de la position stockée avec la valeur de la position cherchée et en remplaçant la plus ancienne des deux valeurs par la plus récente pour la suite du programme.

3 Jeu d'échecs

3.1 Représentation de l'échiquier en machine

Un échiquier dans la réalité est un objet palpable et concret. Il faut donc trouver une représentation pour que la machine puisse suivre le déroulement de la partie au cours du temps.
Il faut pour cela que la machine à un instant donné du jeu ait en sa possession les mêmes informations que le joueur. C'est à dire l'état de toutes les cases de l'échiquier. Chaque case peut être dans l'un des états suivants :

- La case est vide
- La case comporte une pièce blanche (et laquelle)
- La case comporte une pièce noire (et laquelle)

3.1.1 Représentation par un tableau

La première chose que l'on voit est que le plateau de jeu est découpé en 8 rangées de 8 cases. Donc la représentation la plus simple en informatique qui en découle est de définir un tableau à deux dimensions de 8 sur 8. En langage C, on a la déclaration suivante :

```
typeCase tabEchiquier[nbLignes][nbColonnes];
```

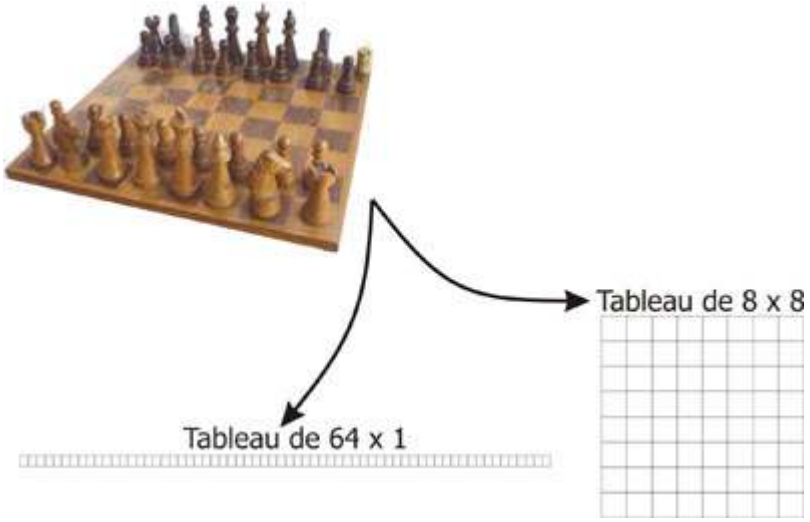
Avec nbLignes=8, nbColonnes=8 et typeCase représentant la structure de notre tableau.

Mais il faut remarquer que la mémoire interne de l'ordinateur est dite linéaire. Ainsi, un tableau à deux dimensions n'est en fait qu'une simplification d'écriture pour représenter un tableau à une seule dimension où les lignes du tableau à deux dimensions seraient collées les unes à la suite des autres.

L'accès à une case du tableau se fait en utilisant la notation tabEchiquier[i][j], i désignant le numéro de ligne, j le numéro de colonne. Le tableau à deux dimensions étant stocké en mémoire sous la forme d'un tableau à une seule dimension, le C calcule le numéro de case correspondant j à partir de i:

```
numeroDeCase = i * nbColonnes + j ;
```

Nous avons donc les deux représentations possibles de l'échiquier suivantes :



Il est donc équivalent d'utiliser un tableau à une ou deux dimensions en terme d'espace mémoire. Cependant, il est peut-être plus simple d'utiliser un tableau à deux dimensions pour simplifier l'écriture du programme. De plus, il est à noter que certains algorithmes de parcours du tableau peuvent être simplifiés ou complexifiés suivant la représentation choisie.

3.1.2 Que stocker dans notre tableau ?

Nous devons stocker dans notre tableau l'état de chaque case comme nous venons de le voir. Mais la manière de stocker cette information peut prendre plusieurs formes. Nous devons sauvegarder les informations suivantes :

- Le type de pièce (pion, cavalier, fou, tour, roi, reine)
- La couleur de la pièce (noire ou blanche)

3.1.2.1 Représenter les pièces avec des enregistrements

Pour cela nous pouvons utiliser une structure :

```
typedef struct {
      T_typeDePiece type ;
      T_couleur couleur ;
} T_case ;
```

avec :

```
typedef enum {PION,CAVALIER,FOU,TOUR,REINE,ROI,CASE_VIDE} T_typeDePiece ;
typedef enum {BLANC,NOIR } T_couleur ;
```

En C, enum définit un ensemble de constantes de type int. Donc chaque case occupera un espace mémoire de deux int.

La place mémoire utilisée pour contenir l'échiquier sera donc de 64 x 2 x sizeof(int), soit **512 octets**. On considère qu'un int vaut 4 octets.

3.1.2.2 Représenter les pièces avec des entiers

A la place d'utiliser une structure, on peut fusionner les informations – type de pièces et de couleurs. En C, le type int peut évidemment contenir plus que 2 x 6 valeurs. Ainsi, on peut arbitrairement définir que les pièces blanches seront représentées par un nombre de 1 à 6 et que les pièces noires seront représentées par un nombre négatif entre –1 et –6. Le type de la pièce sera donc la valeur absolue de l'entier contenu dans la case du tableau, et la couleur sera blanche si entier supérieur à zéro et noir sinon. La valeur zéro représentera une case vide.

La place mémoire utilisée sera donc de 64 x sizeof(int), soit **256 octets**.

3.1.3 Changer pour une représentation différente

3.1.3.1 Les bitboards

Le tableau est la représentation de l'échiquier la plus évidente au premier abord. Mais il existe une autre manière de représenter l'échiquier dans la mémoire de la machine. Au lieu de stocker l'information « sur chaque case de mon échiquier, il y a ou non une pièce », nous pouvons tout à fait garder l'information « les pièces sont sur quelles cases de l'échiquier ».

Il est possible d'utiliser des tableaux binaires de 64 bits appelés « **bitboards** » afin de représenter des prédicats. Chaque bit sera ainsi la réponse à une question posée pour chaque case de l'échiquier. Par exemple le prédicat « où est le roi blanc » serait représenté par le nombre binaire 64 bits avec que des zéros et un seul bit à un pour la case où se trouve le roi blanc sur l'échiquier. Partant de ce principe, il suffit de 12 bitboards pour représenter l'ensemble de l'échiquier :

1. Où est le roi blanc ?
2. Où est(sont) la(les) reine(s) blanche(s) ?
3. Où est(sont) le(les) fous(s) blanc (s) ?
4. Où est(sont) la(les) tour(s) blanche(s) ?
5. Où est(sont) le(les) cavalier(s) blanc(s) ?
6. Où est(sont) le(les) pion(s) blanc(s) ?

Et bien évidemment 6 autres identiques pour les noirs.

La place mémoire utilisée sera donc de 12 x 64/8, soit **96 octets**.

3.1.3.2 Avantages de cette solution

L'utilisation des bitboards est particulièrement performante avec les processeurs 64bits, du fait qu'il était alors possible de représenter l'information par un mot mémoire, traité en un seul cycle d'horloge par le processeur.

La majorité des processeurs actuels sont encore 32 bits cependant, il est possible d'émuler le travail avec des mots de 64 bits de façon transparente pour les programmeurs.

Prenons un exemple concret pour illustrer la puissance de cette solution.

Nous souhaitons connaître si la reine blanche met le roi noir en échec. Avec une représentation par tableau, il faudrait faire les étapes suivantes :

- Retrouver la position de la reine blanche dans le tableau par une recherche linéaire dans le tableau (au plus 64 tests)
- Examiner toutes les cases où la reine peut bouger, et ce dans les 8 directions, jusqu'à ce qu'on ait trouvé le roi noir ou que l'on soit arrivé au bout de toutes les possibilités de mouvement sans résultat.

Donc nous voyons que cette recherche est gourmande en temps d'exécution surtout lorsque la reine ne menace pas le roi adverse puisqu'il faut tester tous les mouvements possibles, ce qui est vrai le plus souvent.

Voyons maintenant ce qu'il faudrait faire pour résoudre cette question en utilisant des bitboards.

- Charger en mémoire le bitboard de « la position de la reine blanche »
- Utiliser ce bitboard comme index dans la table où sont stockés les mouvements précalculés, pour récupérer le bitboard des cases menacées par la reine.
- Faire un ET logique avec le bitboard de la position du roi noir.

Si le résultat est non nul, alors la reine blanche menace le roi noir. Si les trois bitboards utilisés sont déjà dans la RAM, alors l'opération ne nécessite que 3-4 cycles d'horloges !

Prenons un autre exemple, tentons de générer les mouvements du roi blanc. Il suffit de trouver dans la table des mouvements possibles le bitboard correspondant à la position actuelle du roi (table générée une fois pour toutes dans la configuration où l'échiquier est vide, donc aucune pièce ne limite les mouvements des autres). La seule limitation au mouvement du roi est qu'il ne peut pas capturer une pièce de sa propre couleur. Il suffit donc de faire un NON ET du bitboard représentant toutes les cases occupées par des pièces blanches. Le bitboard résultant contient uniquement des 1 où le roi peut aller.

3.2 Fonction d'évaluation

3.2.1 Intérêt

Le principe de fonctionnement d'une IA de jeu d'échec est simple : on calcule des suites de coups possibles à partir de la position courante, chaque position obtenue après chaque coup possible est évaluée suivant des critères et on choisit alors le meilleur coup.

Afin de choisir le meilleur coup possible, il est crucial de réaliser une fonction d'évaluation performante. Elle devra ainsi être conçue de manière à représenter au mieux la situation du joueur. S'il existait une fonction d'évaluation parfaite, il ne serait pas nécessaire de prévoir plusieurs coups en avance.

On peut d'ores et déjà estimer que la fonction d'évaluation sera difficile à réaliser d'un point de vue algorithmique. Il faudra être en mesure de modéliser de manière précise et fidèle à la réalité. Il faudra prendre garde à ne pas créer une fonction d'évaluation qui pénalise l'efficacité de l'IA en faisant une mauvaise estimation qui entraînerait de ne pas retenir une solution qui aurait pourtant été adéquate.

3.2.2 Difficultés pour la modéliser

Le problème d'une fonction d'évaluation est que les possibilités de cas à rechercher sont énormes. Ainsi, si on multiplie le temps pris pour une seule évaluation par le nombre de feuilles qu'évalue l'algorithme Alpha-Beta, on peut se rendre compte que le temps de calcul devient très voire trop grand pour que le joueur humain puisse encore prendre du plaisir à jouer.

Trouver le juste milieu entre évaluation et AlphaBeta représente aussi un point difficile. Bien que les deux soient complémentaires, il est impossible de générer un arbre complet compte tenu de la puissance des machines actuelles. Il faut donc que la fonction d'évaluation calcule des cas qui aurait pu être géré par l'algorithme AlphaBeta si la profondeur avait été plus forte.

L'autre difficulté pour réaliser cette fonction réside en le calibrage de la fonction. La pondération des différents cas a posé problème. En effet, certains cas peuvent se révéler plus intéressants que d'autres à certains moments mais le sont moins à d'autres. En outre, il faut aussi veiller de ne pas trop privilégier l'aspect tactique à l'aspect matériel qui demeure malgré tout prépondérant.

3.2.3 L'attribution de coefficients aux pièces

Afin de prendre les meilleures décisions, l'IA a besoin de connaître la valeur stratégique de chaque pièce. Nos différentes sources s'accordaient à attribuer ces valeurs :

Nom de la pièce	Valeur
Pion	100
Cavalier	300
Fou	300
Tour	500
Reine	900
Roi	0

Etant donné que perdre son roi correspond à la fin de la partie, le roi se voit attribué une valeur nulle. Deux joueurs auront ainsi toujours le point commun d'avoir leur roi sur l'échiquier.

La "material balance" correspond à la valeur totale des pièces d'une couleur. Elle permet de savoir quelle couleur a le dessus par rapport à l'autre d'un point de vue purement comptable. Le roi ne rentre pas dans ce total car sa valeur n'aide pas à différencier les deux totaux.

Material Balance = $\sum(Valeur_{pièce} * Nombre_{pièce})$

Se contenter d'une fonction d'évaluation statique se basant uniquement sur la valeur de pièces serait insuffisant. En effet, beaucoup d'autres paramètres entrent en jeu dans la réflexion d'un joueur : position des pièces, contrôle du centre, etc…

3.2.4 Optimisations de la fonction d'évaluation

3.2.4.1 Mobilité et territoire

Veiller à conserver une grande mobilité des pièces du jeu permet de conserver le maximum de possibilités pour les coups suivants et donc de préserver un maximum de chance de remporter la partie. C'est pourquoi les positions suivantes devront être pénalisées :

- position où des pièces de valeur supérieure aux pions seraient bloquées dans leur mouvement ;
- position où le cavalier perd une partie de sa mobilité car il est trop près d'un bord.

Les positions suivantes se verront octroyées un bonus :

- position où les fous sont placés sur des diagonales ouvertes ;
- position où les tours sont placées sur des colonnes ouvertes ;
- position où les tours sont doublées sur les colonnes ouvertes.

Une autre priorité quand on joue aux échecs est de préserver le contrôle la zone du central de l'échiquier. Une solution serait d'accorder un bonus de valeur à toute pièce se situant au centre. En outre, il faudrait aussi favoriser la « centralisation » des cavaliers.

On pourrait étendre cette fonctionnalité en attribuant une valeur à chaque case de l'échiquier. Plus la case en question serait proche du centre, plus son bonus de valeur sera fort. On considèrera également qu'un joueur contrôle une case si le nombre de ses pièces attaquant cette case est plus grand que le nombre des pièces adverses attaquant cette même case.

3.2.4.2 Préservation des pièces dites importantes

La victoire passe par la défense de son propre roi. Ainsi, les deux mouvements de roque du roi seront bonifiés en valeur tandis que toute intention de déplacer des pions visant à affaiblir le roc sera pénalisée.

Le tropisme représente la pression à laquelle est soumis le roi par les pièces adverses. Il peut s'agir par exemple d'une tour adverse qui mettrait en échec le roi si la pièce située entre le roi et la tour venait à bouger ou se faire prendre par la tour. Ainsi, plus la valeur du tropisme sera élevée, plus le roi se trouvera en danger. Démarrant à zéro, toute valeur de tropisme non nul sera un malus. Il sera aussi possible d'adapter ce principe de calcul aux autres pièces.

Ainsi, il faudra commencer à faire le jeu et attaquer avec des pièces dites mineures (cavalier, fou, pions) afin de préserver les tours et la reine pour le reste du déroulement de la partie. Il sera néanmoins accordé des bonus aux combinaisons favorisant la survie des deux fous ou la conservation du fou de la « bonne couleur » (un fou est dit de la bonne couleur s'il peut attaquer les cases où sont bloqués les pions de l'adversaire).

3.2.4.3 Formations de pions

La maîtrise des pions, bien qu'ayant une valeur très faible individuellement, permet en général de s'approcher de la victoire. Certaines combinaisons de pions devront être pénalisées tandis que d'autres seront bonifiées.

Les positions pénalisées seront :
- pion doublé ou triplé : plusieurs pions sur la même colonne se gênent dans leur mouvement ;
- pions bloqués : deux pions se faisant bloquer face à face ;
- pions passés : un pion qui ne peut être que bloqué ou pris par l'adversaire constitue un grand danger car il menace la ligne arrière ;
- pion isolé : un pion qui n'a aucun pion ami sur ses cotés ;
- les 8 pions : avoir ses 8 pions sur la même ligne gène le déplacement.

Sera bonifié toute intention visant à rapprocher un pion de la dernière ligne arrière de l'ennemi afin d'y obtenir une dame. Le bonus sera d'autant plus grand que la distance sera faible.

3.2.4.4 Réduction du nombre de possibilités à évaluer

Etant donné qu'on ne bouge qu'une pièce par coup, il paraît logique qu'un coup ne nécessite pas forcément d'évaluer une nouvelle fois les positions de toutes les autres pièces de l'échiquier. Une telle idée permettra ainsi de réduire le nombre de calculs de positions par coup calculé. Cependant, il faudra concevoir un module qui définira pour quelles pièces de l'échiquier il faudra recalculer ou non le coefficient.

Il est également possible de réduire le temps d'exécution de la fonction d'évaluation en réduisant le nombre de coups qu'elle a à évaluer. Ainsi tous les mouvements inutiles (ex : aller-retour d'une même pièce) seront à identifier et à bannir. En outre, les coups qui ne viseront rien d'autre que d'immobiliser les pièces adverses seront également exclus. En effet, une telle tactique ne conduirait qu'à étaler ses forces et rendre sa propre défense vulnérable.

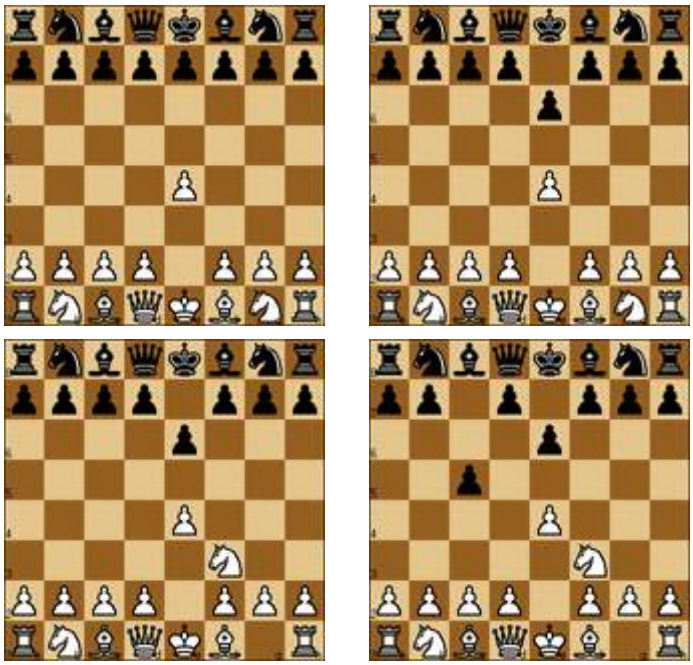
Enfin, si jamais le roi se retrouve en positon d'échec, les seuls coups évalués seront les coups permettant de sortir le roi de sa situation d'échec. Le programme sera ainsi allégé du calcul d'autres coups superflus.

3.3 Ouvertures

Il existe dans le jeu d'échecs un certain nombre d'ouvertures répertoriées. Il s'agit de séquences connues que l'on doit systématiquement utiliser en début. Dans le cas contraire il est fort probable que le joueur se retrouve en une position inférieure par rapport à son adversaire. Une bibliothèque des ouvertures comporte plus de 2000 pages.
Les ouvertures ne sont pas stockées sous forme de séquence de coups, mais sous forme de positions indexées afin de ne pas être trompé par une inversion de coup. Cela signifie que l'ordinateur repère une ouverture non pas par une séquence de coups joués par son adversaire, mais par l'emplacement des pièces sur l'échiquier. L'ordinateur peut ainsi savoir quel coup il doit jouer en réponse.

Exemple d'ouverture :

1. e2-e4 e7-e6
2. g1-f3 c7-c5



Face à cette position l'ordinateur (blanc ici) joue en d4. Dans le cas contraire l'ordinateur n'a pas reconnu l'inversion de coup qui a transformé une partie française en sicilienne. En effet la séquence classique pour atteindre la position dite sicilienne est :

1. e2-e4 c7-c5
2. g1-f3 e7-e6

Ne trouvant pas cette séquence standard, l'ordinateur est incapable de reconnaître une position sicilienne. L'ordinateur est perdu s'il stocke ses ouvertures sous forme de séquences et non sous forme de position indexée.

3.4 Fermetures

Par leur complexité stratégique, les finales ont longtemps été le point faible des jeux d'échecs. Grâce à leur force de calcul et à leurs algorithmes de recherche efficace, les ordinateurs sont performants en milieu de partie. Mais dans un final, l'aspect stratégique est bien plus important et ne peut pas être pris en compte dans un algorithme utilisant les arbres de jeu comme les algorithmes de type MiniMax ou α-β.

Prenons un exemple simple : une finale Roi et Pion contre Roi et Pion. La situation est la suivante :

La partie semble tourner à l'avantage du joueur blanc. En effet le pion noir étant bloqué par le roi blanc, les noirs sont dans l'incapacité de remporter la partie. Considérons que l'ordinateur soit le joueur blanc. En utilisant un algorithme de recherche de coup du type α-β à une profondeur de moins de 6 coups, l'ordinateur va considérer qu'il doit prendre le pion en h2. En effet, l'ordinateur se presse de capturer le pion car si l'adversaire bouge son roi en g3, il sera dans l'incapacité de capturer le pion au tour suivant. Il y a donc un manque à gagner pour l'ordinateur. De plus l'intelligence artificielle ignore que son pion en a2 peut se transformer en dame en 6 coups car la profondeur de son arbre de jeu n'est pas suffisante.

Il est donc nécessaire d'inclure des méthodes d'analyse pour les fins de jeux pour faire mat en un minium de coup. Il faut parfois savoir sacrifier le court terme pour le long terme. Pour reconnaître certains types de positions particulières, les jeux d'échec utilisent des tables de fin de jeu dites tables de fermeture, stockant les coups à jouer pour faire mat. Actuellement le problème des finales aux échecs est résolu avec 5 et 6 pièces (sans pions) au maximum.

Il est possible de générer ces tables. La première étape consiste à définir toutes les positions possibles des pièces restantes sur l'échiquier. Pour une finale du type KNNK, il existe $64 \times 64 \times 64 \times 64 = 16777216$ positions possibles au maximum. On peut soustraire à ce chiffre les positions impossibles et les symétries éventuelles. A partir de cette liste, on extrait toutes les situations possibles où un des rois est mat, puis on calcule à partir de cette liste de situation, les coups qui conduisent à un mat en un coup, puis en deux, puis en n coups… Une fois la situation initiale retrouvée lors de la recherche des coups dans le sens inverse, on obtient ainsi une liste de coups à effectuer pour mettre mat l'adversaire. On appelle cette méthode, l**'analyse rétrograde**.

En 1991, Larry Stiller a construit la plus longue finale gagnante connue à ce jour. La résolution complète demande 223 coups. Seul un ordinateur peut réaliser un tel exploit.

3.5 Algorithme et contrôle du temps

Lors d'une partie d'échecs, chaque joueur dispose d'un temps total pour l'ensemble de la partie. Un ordinateur doit donc être capable de gérer son temps au cours de la partie. Il existe principalement deux techniques pour que l'ordinateur prenne en considération cette contrainte temporelle. La première consiste à accorder à l'ordinateur une durée maximale de réflexion et d'estimer ensuite la profondeur à laquelle l'IA du jeu peut aller sans dépasser la durée fixée. Cette première technique pose un problème important : comment réaliser une fonction évaluant correctement la profondeur possible par rapport à la durée maximum de réflexion ? Il est difficile de concevoir une telle fonction.

Une deuxième technique plus optimisée consiste à évaluer une première version de l'arbre à faible profondeur et d'affiner ensuite la recherche en augmentant progressivement la profondeur si le temps de réflexion maximal n'est pas encore atteint. De plus cette technique ("Depth First Iterative Deepening") permet un meilleur parcours par les algorithmes de type algorithme α-β. En utilisant les évaluations précédentes, l'algorithme α-β peut examiner les sous-arbres de façon plus optimisée en parcourant d'abord les meilleurs coups permettant ainsi un meilleur élagage.

3.6 Algorithme génétique

Inspiré de la théorie sur l'évolution de Darwin, les algorithmes génétiques sont apparus pour simuler des phénomènes évolutions. Selon Darwin, les modifications des gènes chez les êtres vivants ne se faisaient pas de manière totalement aléatoire. Des facteurs environnementaux favorisaient l'apparition ou le maintien de certains attributs génétiques. Ainsi, tout individu né avec des caractéristiques génétiques ne lui permettant pas de s'adapter à son environnement venait à disparaître sans laisser de descendance. Les individus dotés des gènes favorisant leur adaptation survivait et transmettaient leurs gênes à leurs descendances.

Concevoir des IA en partant d'algorithmes génétiques diffère de l'approche classique. En effet, selon l'approche classique, on modélisait un système à partir des contraintes définies. Concevoir des algorithmes génétiques revient à modéliser les contraintes de l'environnement et de laisser le système évoluer par lui-même de manière autonome en s'adaptant aux contraintes de l'environnement. Par exemple, si on voulait modéliser un être humain selon des algorithmes génétiques, les contraintes seraient la modélisation des os, des muscles, des tendons ainsi que la gravité. L'humanoïde serait incapable de tenir debout au départ mais au fur et à mesure de ses propres essais, il sera alors capable de marcher, courir et enrichir sa panoplie de mouvement. Vous pourrez trouver un exemple de cette conception dans l'article paru sur le site du magazine Wired et dont le lien est donné en annexe.

Modéliser l'IA d'un jeu avec des algorithmes génétiques paraît très attractif. En effet, à partir des contraintes à définir qui ne sont en fait que les règles du jeu d'échec, l'IA serait en mesure d'apprendre de ses défaites et de se s'améliorer au fur et à mesure des parties. Cependant, le choix d'une telle méthode se révélerait inadaptée à notre problème. S'il ne fait aucun doute que l'IA saura rapidement déplacer toutes ses pièces sur l'échiquier, il est moins évident de penser qu'elle saura devenir un adversaire redoutable. Par son approche empirique du jeu, l'IA ne procédera que par identification et répétition de séquences apprises au cours de ses parties précédentes. On peut ainsi dire que paradoxalement, une IA génétique montrerait moins de faculté d'adaptation face à son adversaire qu'une IA composé d'un algorithme linéaire de parcours d'arbre et d'une fonction d'évaluation.

4 DiamondChess : les solutions utilisées

Afin de tirer partie au mieux du peu de temps qui nous était imparti – en effet, la programmation d'un jeu d'échec requière de longues étapes d'ajustement notamment sur la fonction d'évaluation – nous avons voulu nous consacrer au maximum sur l'ia du jeu et voir de près ces petits détails.

4.1 Représentation en mémoire - l'abandon des bitboards

Bien que l'étude préliminaire ait montré que les bitboards sont les moins gourmands en mémoire, leur implémentation nous a posé problème et nous avons du revenir à une représentation plus classique du jeu en mémoire. Nous arrivions avec les bitboards, en quelques opérations simples, à répondre à des questions comme « le roi peut-il bouger ? ». Mais dès qu'il s'agissait de trouver les mouvements d'une tour, le problème restait entier. En effet, la tour peut se déplacer de plusieurs cases contrairement au roi. Et il faut, pour vérifier si le déplacement est possible, vérifier qu'il n'y a aucune pièce entre la position initiale et la position finale de la pièce. N'ayant pas trouvé de façon simple d'utiliser les bitboards pour répondre à cette question, nous utilisions les bitboards à la manière d'un tableau d'entier, mais en effectuant plus de calculs que pour un tableau classique.

Pour économiser du temps de calcul, nous avons changé la représentation quitte à rendre le programme plus gourmand en mémoire. Les phases de test ont montré que le taux d'utilisation de la mémoire reste acceptable pour ce type d'application (autour de 10 Mo).

4.2 Algorithme de recherche utilisé

Nous avons opté pour un algorithme alpha-beta, nos recherches nous ayant montré qu'il donnait de très bons résultats, et qu'il était un très bon compromis entre rapidité et nombre de noeuds analysés.

4.3 Fonction d'évaluation

a) On vérifie si le roi est en situation d'échec, de mat ou s'il y a pat ;
b) Calcul de valeur de la position :

MB_{IA} : valeur statique totale des pièces appartenant à l'intelligence artificielle
MB_{AD} : valeur statique totale des pièces appartenant à l'adversaire
AT : Evalution de l'avantage Tactique qui est calculé avec pour référentiel l'IA

- AT > 0 s'il y avantage pour l'IA
- AT < 0 s'il n'y avantage pour l'adversaire

$$E = MB_{IA} - MB_{AD} + AT$$

c) On transmet les valeurs retournées par la fonction d'évaluation à l'algorithme de parcours de l'arbre.

La principale faiblesse de l'attribution de bonus / malus réside dans le fait que les priorités en début de jeu ne sont pas les même en milieu voire fin de jeu. Par exemple, l'intérêt qu'il y à augmenter le coefficient d'importance des cases du centre perdra de son importance au fur et à mesure. En effet, moins il y aura de pièces restantes, plus il faudra concentrer ses attaques vers le roi et non plus le centre.

4.3.1 Procédure de calcul

a) Le calcul de la Material Balance :

Le calcul de la Material Balance se calcule au fur et à mesure de chaque coup. Ainsi, à chaque prise, la MB est mise à jour pour la couleur concernée. Cela nous permet de faire des économies de cycle de calcul car on ne recalcule pas la MB en parcourant le tableau inutilement à chaque fois.

b) Le calcul de l'avantage tactique :

Voilà la démarche suivie :

```
EvaluerAvantageTactique(couleurIA)
Entrées : la couleur de l'IA
Sorties : l'évaluation
{
	On regarde la situation au centre et on attribue un bonus à qui le
	domine.

	Pour toutes les pièces présentes sur l'échiquier
	{
		On regarde si cette pièce se trouve encore en position initiale.
		On octroie un malus si c'est le cas.

		On regarde si la pièce est à la fois isolée et menacée, ou si
		elle bénéficie d'un nombre de protections inférieur à celui des
		menaces.
		Si c'est le cas, on retire la valeur de cette pièce à
		l'évaluation finale.
		Sinon on octroie un bonus par rapport au nombre de protections
		et on attribue des bonus pour les cas de protection réciproque.

		On évalue ensuite la mobilité de la pièce.
		Plus de possibilités sont offertes, plus le bonus est grand
		On cherche quelle pièce on peut attaquer et attribuer un bonus
		supplémentaire :
			S'il s'agit d'une attaque multiple (« fourchette »)
			S'il s'agit d'une attaque à sens unique
			On étudie le tropisme
		On favorise également les possibilités de roque.
	}
}
```

4.4 Table d'ouvertures

Nous avons implémenté des tables d'ouvertures afin que le programme ne tombe pas dans les pièges qu'un simple alpha-beta ne pourrait pas discerner. La table utilisée – au format Access – regroupe des centaines d'ouvertures connues. Dans certaines configurations, l'ordinateur sait encore quoi jouer après 12 demi-coups.

Pour que l'utilisateur puisse apprendre les noms des ouvertures, une option permet de visualiser sur l'échiquier les coups possibles présents dans la table des ouvertures ; un survol d'une de ses cases à la souris entraîne l'affichage du nom de l'ouverture entre les deux compteurs.

4.5 Tables de fermetures

Les tables de fermetures les plus utilisées sont les tables de Nalimov compressées. Ces tables sont réparties sur plusieurs fichiers en fonction des pièces restant en fin de partie. Les tables les plus couramment utilisées sont celles à deux, trois et quatre pièces. Elles contiennent la « distance au mat ». C'est-à-dire que pour une position de jeu donnée, il faut procéder de la façon suivante :

- Initialiser les tables de fin de jeu
- Créer à partir de la position actuelle de l'échiquier (à partir de notre représentation) une chaîne de caractères formatée compréhensible par la fonction de Nalimov.
- Utiliser l'algorithme de Nalimov pour lire dans les tables la « distance au mat ». C'est-à-dire le nombre de coups à jouer avant de faire mat.

Ainsi, pour tous les coups générés à une position donnée, on interroge autant de fois que nécessaire les tables de fin de jeu. On joue le coup qui mène le plus rapidement au mat.

L'algorithme permettant d'interroger les tables compressées étant très complexe et peu commenté, nous n'avons pas réussi à l'implémenter.

4.6 L'historique et la sauvegarde des parties

Nous avons mis en place un historique des coups joués – affiché sur le coté droit de l'échiquier – afin de permettre au joueur de visualiser le déroulement de la partie mais aussi pour que nous puissions vérifier ce que joue l'IA, notamment pendant les phases de tests. L'utilisateur peut restaurer ou annuler une série de coups.

De plus, nous avons ajouté la possibilité de sauvegarder les parties afin de pouvoir tester notre IA plusieurs fois de suite à un problème donné.

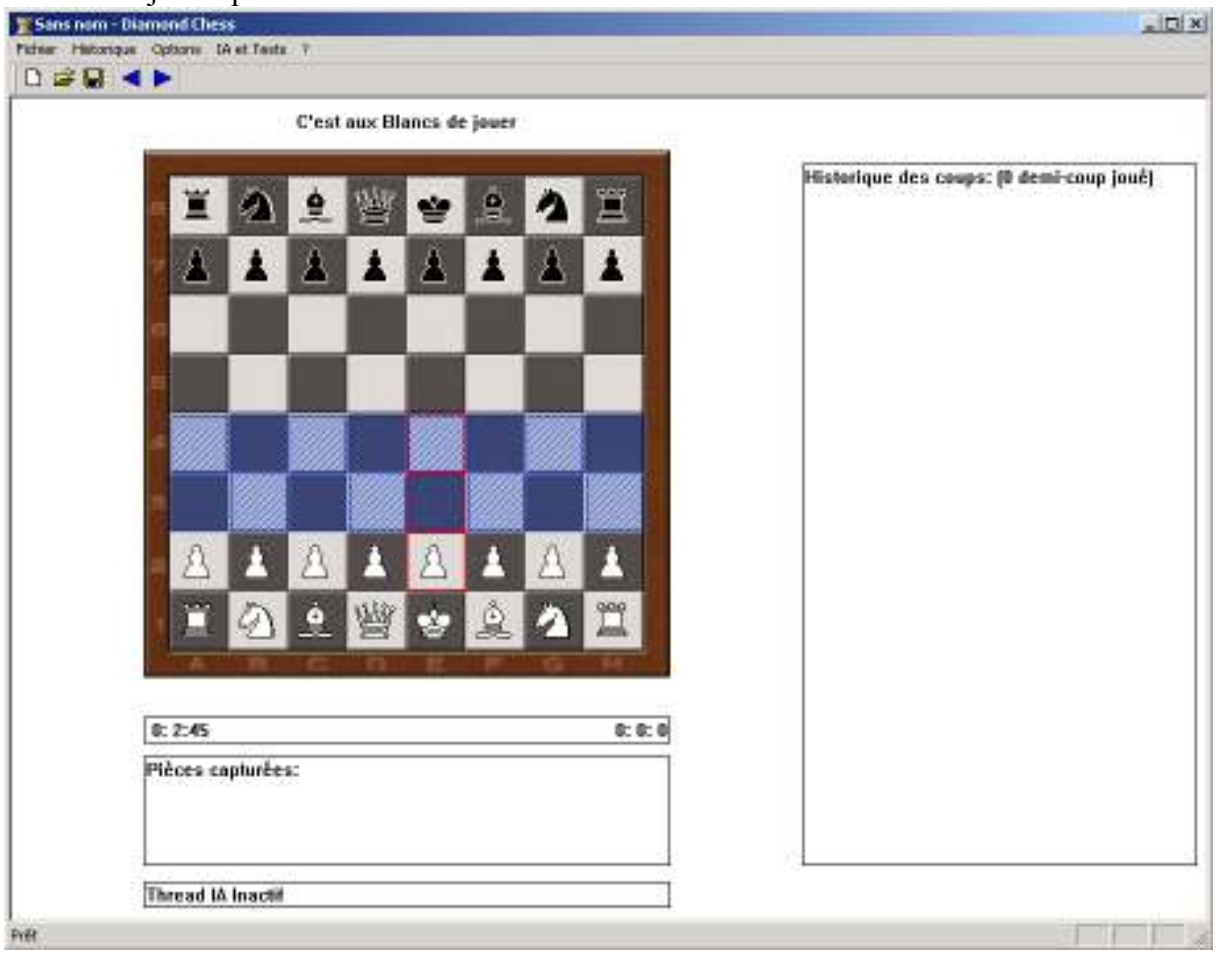
4.7 Programmation multi-thread

L'IA nécessite un temps de calcul long avant de trouver quel coup jouer. Sans thread, le programme semble figé et plus aucune action ne peut être effectuée (ouvrir un menu par exemple) tant que l'IA n'a pas terminé sa réflexion. Nous avons donc mis en place un thread de travail dans lequel on lance l'IA. Quand elle a fini de jouer, elle envoi un message au thread principal afin de redessiner la fenêtre et de donner la main au joueur.

Cette solution permet de continuer à incrémenter le compteur de l'IA pendant qu'elle réfléchit.

5 Mode d'emploi

L'écran de jeu se présente comme suit :



Décrivons l'écran de jeu.

- En haut, au-dessus du plateau de jeu : le tour de jeu afin de savoir qui doit jouer.
- A droite du plateau de jeu : l'historique des coups.
- Sous l'échiquier : les deux compteurs indiquent le temps de jeu de chaque joueur. En cas de survol d'une case bleue, le nom de l'ouverture connue dans la table d'ouverture s'inscrit entre les deux compteurs
- En dessous les pièces capturées par chaque joueur.
- Et encore en dessous, une indication décrivant l'état de l'IA.

Code des couleurs utilisées :

- Les cases bleues représentent les coups possibles présents dans la table d'ouverture.
- Quand le joueur sélectionne une pièce de sa couleur, la pièce est entourée d'un liseré rouge. Toutes les cases où la pièce peut se déplacer sont également entourées de rouge.

5.1 Exemple de partie

Exemple de partie IA contre IA en profondeur 4 :

6 Conclusion

L'analyse théorique nous a montré la complexité de réaliser une intelligence artificielle pour un jeu d'échecs. Pour obtenir un programme capable d'obtenir la victoire face à un humain, il ne "suffit pas" uniquement d'avoir une vision du jeu à plusieurs coups en profondeur. En effet, une IA aux échecs doit être capable d'établir une stratégie lors des finales de jeu et de savoir évaluer une position de l'échiquier à sa juste valeur. La fonction d'évaluation est l'un des points-clés de la programmation d'une IA d'un jeu d'échecs.

Le projet d'initiative personnel nous a donné l'opportunité, par le biais d'un sujet intéressant et d'une réalisation concrète, d'approfondir la théorie des jeux qui nous a été introduite en cours.

Grâce à ce projet nous sommes passés d'un monde à un autre : du statut de simple joueur nous sommes devenus concepteurs de jeu. Après des heures d'efforts nous avons enfin pu nous mesurer à notre création. Il est réjouissant de voir que notre IA sait nous tenir tête. Pour encore améliorer l'intelligence du programme, il serait nécessaire de jouer de nombreuses parties contre l'IA afin d'affiner au mieux la fonction d'évaluation.

En plus de notre problématique d'intelligence artificielle, nous avons été amenés à modéliser et concevoir un programme complet. La réalisation de ce projet représente un lourd investissement en temps étalé sur la période d'octobre à avril. Le travail en équipe a été un net avantage tout au long de ce projet. En effet au cours de la phase de recherche nous nous sommes mutuellement expliqué les algorithmes ce qui nous a permis de dégager les points les plus délicats et de creuser plus en détails. Ensuite au cours de la phase de modélisation, nous nous sommes repartis le travail et nous nous sommes entre-aidés pour avancer et aboutir à ce jeu d'échecs.

7 Bibliographie & Webographie

7.1 Bibliographie

- *Artificial Intelligence Fourth Edition*, G. F. LUGER
- *Intelligence Artificielle & Informatique Théorique 2ème Edition.* J.-M. ALLIOT, T. SCHIEX, P. BRISSET, F. GARCIA

7.2 Webographie

- http://www.ffothello.org/info/algos.htm
- http://www.ifrance.com/jeudechecs/pageechecs.htm
- http://chess.verhelst.org/search.html
- http://www.paginus.com/lesechecs/a.html
- http://wannabe.guru.org/scott/hobbies/chess/
- http://www.gamedev.net/reference/list.asp?categoryid=18
- http://www.chessopolis.com/cchess.htm#info
- http://www.aarontay.per.sg/winboard/egtb.html (Endgame Table Guide)
- http://www.wired.com/wired/archive/12.01/stuntbots.html?pg=1&topic=&topic_set=