Le Jeu de QUARTO

— Etude de méthodes de recherche arborescente

Par

LIANG Ke

MASTER 2ème en Informatique

sous la direction

de

Bruno BOUZY

INTRODUCTION..1

I.Jeux de Reflexion et Quarto..3

I.1.L'Etat de l'Art..3

I.2.Quarto..6

I.3.L'Etat de l'Art pour Quarto..8

II.Technique de la programmation de jeu 9

II.1. Technique classique..9

II.1.1. MinMax 9

II.1.2. AlphaBeta 10

II.1.3. Autres Améliorations..11

- Itérativement Deepenning..11
- Table de transposition..12

II.2. Technique Monte-Carlo..13

- Sélection..13
- Expansion..14
- Simulation..14
- Retour 14

III.La Réalisation du programme..15

III.1. L'objectif et la conception..15

III.2. Les objets et l'architecture...15

III.3. La programmation 16

III.4. Amélioration de la programmation..29

IV.Expérience et Résultat...32

IV.1. StochaEngine *S*...32

IV.2. MinMaxEngine *M*...33

IV.3. AlphaBetaEngine *A* 34

IV.4. *TA,IA* et *ITA*...35

IV.5. UCT *U*..36

IV.6. AlphaBetaEngineUCT *UA*...37

CONCLUSION 39

BIBLIOGRAPHIE SITOGRAPHIE..41

ANNEX..43

INTRODUCTION

L' Intelligence Artificielle (IA) est la partie de l'informatique dont le but est d'élaborer des systèmes intelligents, c'est-à-dire des systèmes qui possèdent les caractéristiques que nous associons avec l'intelligence dans le comportement humain: l'apprentissage, le raisonnement, la solutionnement de problèmes, etc. Le jeu de réflexion est l'une des premières tâches à laquelle l'IA s'est attelée. Le terme “jeu de réflexion” est employé dans le sens où des chercheurs en IA s'intéressent aux jeux dans lesquels une réflexion est utile pour bien jouer à ces jeux. Cela regroupe les jeux comme les Echecs, le GO, Othello Amazons etc. Ces jeux sont intéressants pour des cherheurs parce qu'ils sont trop difficiles à résoudre. Pour certains jeux, l'arbre de recherche est trop large à explorer. Jusqu'à présent, on a eu plusieurs méthodes efficaces pour la recherche arborescente, comme *MinMax, AlphaBeta, Itérative Deepenning* etc. Elles ont aidé à augmenter le niveau de jeu, à tel point que les machines ont surpassé les humains aux dames et à Othello, et qu'elles ont battu des champions d'échecs et de backgammon et qu'elles sont compétives dans de nombreux autre jeux.

Mon stage, proposé par le professeur Bruno BOUZY, est basé sur un jeu de réflexion qui s'appelle Quarto. Le travail est sur la programmation de ce jeu en appliquant des méthodes de recherche arborescente classique , et aussi *UCT* (une nouvelle méthode stochastique qui marche très bien dans le jeu GO).

Ce stage m'a beaucoup attiré parce que j'aimerais avoir des connaissances sur les techniques pour programmer un jeu , et j'aimerais aussi pouvoir réaliser un jeu où la machine peut atteindre un haut niveau pour surpasser l'humain. Dans le premier semestre de Master 2, j'ai déjà fait un projet sur le jeu Clobber. J'ai programmé en utilisant des méthodes *MinMax* et *AlphaBeta*. Donc pour avoir plus de connaissances sur les méthodes et ses améliorations, j'ai décidé de faire ce stage.

Pour ce stage, je vais programmer ce jeu en C++, sous le système Linux, en utilisant différentes méthodes comme: *MinMax, AlphaBeta, Itérative Deepenning, Table de Transposition, stochastique, UCT*, et aussi des méthodes mélangées. Ce program me perrmet de:

1. Jouer à deux joueurs.
2. Jouer à un joueur contre la machine .
3. Jouer à deux machines (Matériellement le programme est lancé sur une machine, mais peut simuler le jeu entre deux machines.)

L'objectif de ce stage est de comparer ces méthodes, d'analyser les résultats du

programme , et de trouver des méthodes qui marchent très bien dans le jeu Quarto.

Dans la première partie du rapport, je vais présenter l'état de l'art des jeux de réflexion(Go, Echec, Checkers etc), la définition et la règle du jeu Quarto, Quarto n'est pas aussi connu que les jeux classiques , mais il présente quelques caractères assez intéressants. J'ai trouvé des travaux des autres personnes et j'ai aussi communiqué avec eux pour échanger des expériences.

Dans la deuxième partie, je vais parler des techniques pour programmer un jeu: la technique classique qui est basé sur l'algorithme *MinMax*, et la techinque Monte-Carlo qui passionne des chercheurs en ce moment. Ce sont des notions préalables pour ma programmation sur Quarto.

Ensuite je vais notamment présenter les travaux pratiques sur la programmation. La programmation est réalisé en C++, sous le système linux. L'objectif de mes travaux est de réaliser les techniques, et trouver la méthode meilleure. Il y a 8 méthodes que j'ai réalisées: *StochaEngine, MinMaxEngine, AlphaBetaEngine, IDAlphaBetaEngine, TTAlphaBetaEngine, IDTTAlphaBetaEngine, UCTEngine* et *AlphaBetaEngineUCT.* En outre je vais interpréter d'autres objets de mon programme et des amélioration.

Enfin, la quatrième partie présente l'interprétation des résultats en comparant les différentes méthodes. Par exemple, en utilisant ce programme, je peux tester 100 ou plus parties, jouées entre une machine qui utilise la méthode *AlphaBeta* de profondeur 6 et une machine qui utilise la méthode *UCT* au paramètre de 30 secondes. Le programme permet aussi de comparer toutes les méthodes entre elles.

Pour conclure, c'est un stage intéressant. Le jeu Quarto est intéressant parce qu'il est très variable en raison de sa règle particulière, même si le tablier n'est pas très grand, et même si les pions ne sont pas nombreux. Quant aux méthodes, on peut voir leurs différences, leurs avantages, et leur inconvénients. Pour Quarto, une méthode mélangé (*AlphaBeta +UCT*) est toujours plus forte que les autres méthodes que j'ai expérimentées.

I. Jeux de Reflexion et Quarto

Les jeux de réflexion sont un terrain d'étude priviliégié pour l'intelligence artificielle. Ils ont toujours fasciné les informaticiens. Dès 1945, au tout début des ordinateurs programmables, Konrad Zuse (inventeur du premier ordinateur programmable et du premier langage de programmation) s'est intéressé aux échecs, suivi par Claude Shannon (inventeur de la théorie de l'information), Norbert Wiener (fondateur de la cybernétique) et Alan Turing.

Pourquoi cette fascination des informaticiens pour le domaine des jeux? Tout d'abord parce qu'il semble intuitivement (et inconsciemment) que la pratique de jeux intellectuels soit le propre de l'homme. De plus, tout jeu apparaît un peu comme un champ clos de tournoi au Moyen Âge: chacun est astreint à respecter des règles strictes et précises, et ne peut l'emporter que grâce à sa valeur et à son habileté. La victoire et la défaite sont sanctionnées clairement et sans appel possible. Réaliser une machine qui joue aussi bien qu'un homme, voire mieux, prouverait l'intelligence des ordinateurs, et serait apparue comme une grande réussite pour l'IA, du moins à ses débuts.

Dans la suite, je vais notamment présenter l'état de l'art des jeux de réflexion(Echec, Othello, Jeu de Dame, Bridge et GO), et aussi le jeu Quarto sur lequel je travaille.

I.1 L'Etat de l'Art

- **Echecs**
  Au XIX e siècle, Charles Babbage(1792-1871) , mathématicien britanique et l'un des précurseurs de l'informatique, avait envisagé de programmer sa machine analytique pour jouer aux échecs. Après en 1950, Claude Shannon décrivit les rudiments d'un programme d'échecs, comme le fit également Alan Mathison Turing. 47 ans plus tard, le programme *Deep Blue*, développé par Murray Cambell, Feng-Hsiung Hsu et Joseph Hoane chez IBM, gagnait contre Garry Kasparov un match d'exhibition en six parties. Le vainqueur était un ordinateur parallèle composé de 30 processeurs IBM RS/6000 exécutant la ' *recherche logicielle*' et de 480 processeurs VLSI dédié qui prenaient en charge la génération des coups, la '*recheche matérielle*' pour les derniers niveaux de l'arbre et l'évaluation des noeuds feuilles. Le programme générait jusqu'à 30 milliards de positions par coup et atteignait régulièrement une profondeur de 14. C'était un succès au niveau du matériel, mais les créateurs de *Deep Blue* ont fait remarquer que les

extensions de recherche et la fonction d'évaluation étaient également cruciales.

Après en 2002, le progrmme *Deep Fritz*, développé par Frans Morsch et son ami Ed Schroder, a fait match nul avec le champion du monde classique en titre Vladimir Kramnik. Les conditions du match étaient plus favorables au joueur humain et le matériel employé n'était qu'un PC ordinaire au lieu d'un superordinateur comme *Deep Blue*. Kramnik n'en a pas moins déclaré: " Il est désormais évident que le meilleur programme et le champion du monde sont à peu près du même niveau." Du 25 novembre au 5 décembre 2006, *Deep Fritz* jouait un match en six manches contre Kramnik à Bonn. Cette fois-ci *Deep Fritz* a gagné sur le score de 4–2, dont un point perdu sur un gaffe impensable de la part de Kramnik.

- **Checkers**
  Le premier programme pour les checkers est celui d'Arthur Samuel en 1952. Et le programme actuellement le plus fort est *CHINOOK* (Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu and Duane Szafron) de 1990. En 1994 il a battu Marion Tinsley qui a été champion du monde depuis 40 ans. Donc *CHINOOK* est le premier ordinateur qui est considéré comme un championnat du monde officieux.

  Jonathan Schaeffer estime que, dans l'état actuel de la technologie, il doit être possible de résoudre complètement le jeu de checkers, c'est-à-dire de construire un programme capable de jouer parfaitement de façon certaine.

- **GO**
  Le Go occupe une place à part : les meilleurs programmes se font (encore) battre par des joueurs amateurs ayant un peu de pratique, parce que il fait partie des jeux extrêmement difficiles à programmer. En effet, l'arbre de jeu ne peut pas être examiné de façon complète, même sur deux niveaux, en raison du très grand nombre possible de coups à chaque tour de jeu (plus de 300 en début de partie). D'autre part, le jeu est avant tout lié à des concepts positionnels stratégiques très difficiles à formaliser simplement.
  Le Go est très étudié en ce moment. Pendant 2000 – 2005 , l'approche Monte-Carlo, qui a été le premier étudié en 1993 par Bernd Brugmann dans le programme *Gobble*, attirait de nouveau aux chercheurs: Bernard Helmstetter (*Oleg*), Tristan Cazenave (*Golois*), Bruno Bouzy (*Indigo*),Gillaume Chaslot (*Mango*), Remi Coulom (*CrazyStone*).

  Actuellement le programme *MoGo*, qui est inspiré de la stratégie de *CrazyStone* de Rémi Coulom et qui est classé numéro 1 depuis août 2006 parmi 142 sur le Computer Go Server, a été développé au sein du projet

TAO en collaboration avec le CMAP de l'Ecole Polytechnique. *Mogo* est actuellement l'un des tout meilleurs joueurs de GO artificiels.

- **Othello**
  Le jeu d'Othello (également connu sous le nom de Reversi) est un jeu de société à deux joueurs. Il est devenu, comme les échecs ou les dames, l'un des grands classiques des jeux de réflexion. Il est certainement plus connu comme jeu informatique qu'en tant que jeu de plateau, parce qu'il est l'un des jeux les plus programmés au monde! En 1997, le programme *Logistello* (Buro 2002), qui est actuellement le meilleur programme mondial, battait le champion du monde humain, Takeshi Murakami, par six parties à rien(6-0). On reconnaît généralement que les humains ne font pas le poids face aux ordinateurs à ce jeu.

  En France, il existe une fédération (Fédération Francaise d'Othello) qui permet de fédérer les clubs français, organise des tournois et compétitions officielles, et participe à des animations pour faire découvrir le jeu d'Othello des plus jeunes aux plus âgés.

- **Bridge**
  C'est un jeu caractérisé par une information imparfaite: les cartes d'un joueur sont dissimulées aux autres joueurs. Le bridge est aussi un jeu à plusieurs joueurs, bien que ceux-ci soient associés sous la forme de deux équipes de deux joueurs. En ce moment , peu de gens se sont intéressés à la programmation du bridge, peut-être parce que la raison de ce déintérêt est l'aspect '*chance*' qui existe au bridge. Les gens pensent que si un programme d'échecs l'emporte, il est meilleur, alors qu'un programme de bridge peut simplement 'avoir de la chance'. Par rapport à la raison technique, la réalisation d'un programme de bridge est beaucoup difficile qu'aux échecs. Il est impossible d'utiliser directement des méthodes de recherche classiques, en raison du très important nombre de distributions possibles de mains.

  Actuellement le programme *GIB* (Ginsberg,1999) a nettement remporté le championnat de l'an 2000. Il utilise la méthode de l'espérance moyenne sur la clairevoyance avec de modifications cruciales.

En plus des jeux dont je parle ci-dessus, il y a encore des autres jeux comme Backgammon, Amazones etc qui sont déjà étudiés. En somme, la table (Table.1) de (Herik & al 2002) peut décrire l'état de l'art de ces jeux .

résolu	*supra-humain*	*champion du monde*	*grand maître*	*amateur*
Tic-tac-toe	Othello	Echecs	Shogi	Go (9x9)
Go-moku	Checkers	Dames (10x10)	Echecs chinois	Go(19x19)
Hex (7x7)	Backgammon		Bridge	Amazones
Awari	Scrabble		Go (7x7)	

Table.I.1

I.2 Quarto

Quarto est un jeu de réflexion pour 2 joueurs. Il est créé par Blaise Muller, primé en 1985 au Concours international de créateurs de jeux de société sous le nom de 4x4 et édité depuis 1991 par Gigamic. En 1992, Quarto est récompensé l'As d'Or Jeu de l'année .

I.2.1) Contenu du jeu

Ce jeu se compose d'un tablier et de16 pions.

- un tablier de 16 cases (4x4)
- 16 pions différentiables par 4 caracéristiques:
    - la couleur – noir / blanc (bois clair / bois foncé)
    - la hauteur – petit / grand
    - le sommet – plein / troué
    - la forme – paralléppipèdique /cylindrique

  Toutes les combinaisons (exemple: grand, noir, troué et cylindrique) sont représentées et chacune n'est représentée qu'une seule fois.

I.2.2) Un tour du jeu

Ce jeu est joué par 2 joueurs tour par tour. Chaque tour d'un joueur peut être composé par 2 parties:

- Le joueur place le pion qui est donné par l'adversaire sur un emplacement libre du tablier.
- Le joueur donne un des pions qui n'a pas été joué à l'adversaire à placer.

I.2.3) Condition de victoire

Dans la règle de base, le but du jeu est d'être le premier de constituer une ligne de quatre pions présentant une caractéristique commune. Cependant, si l'on joue systématiquement de manière défensive, les parties débouchent très souvent sur des positions nulles. Aussi, la règle prévoit d'autres configurations de gain, suivant quatre niveaux de jeu .

- La règle de base correspond au niveau un, la victoire est obtenue en réalisant un alignement.
- Le niveau deux permet de gagner en réalisant un alignement ou un petit carré(par exemple a1,a2,b1,b2 – Figure I.1).

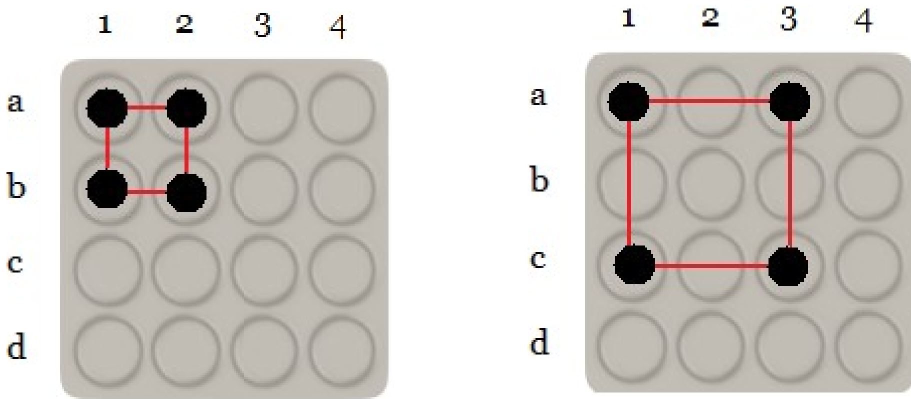

Figure I.1 Figure I.2

- Le niveau trois, en plus des positions de gain du niveau 2 permet de l'emporter en réalisant un carré plus grand(par exemple a1,a3,c1,c3– Figure I.2).
- le niveau quatre, reprenant les positions de gain du niveau trois, y ajoute les carrés 'tournants', à savoir par exemple les cases a2,b1,b3,c2 (Figure I.3), ou, encore plus difficile à voir, les cases a2,b4,d3,c1 (Figure I.4).

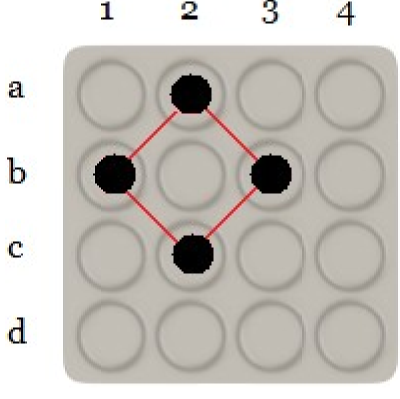

Figure I.3

Figure I.4

I.3 L'Etat de l'Art pour Quarto

Bien sûr Quarto n'est pas aussi connu que les autres jeux de réflexion: Go, Echecs. Il n'y a pas de tournoi officiel entre les machines comme Go, même entre les humains. Mais il a quand même intéressé aux chercheurs pour le programmer. Il existe des jeux Quarto joués sur Internet, mais ils sont réalisés de jouer entre 2 humains. J'ai trouver 2 programmes qui permettent de jouer entre l'humain et la machine.

1)Linéo, Cest un logiciel libre publié sous GPL. Il est possible de jouer seul contre l'ordinateur, à deux sur le même ordinateur ou en réseau via Internet.

Linéo est le projet de l'équipe Natsimhan. Jonathan Buron (Nathan) est à l'origine de ce projet et c'est avec l'aide de Pierre-Yves Bonnefoy (PYB) que le projet se concrétisa pour devenir Linéo. D’autres personnes (Sim, Mimi, Jojosan, Suporel, ...) ont rejoint l'équipe Natsimhan pour améliorer le jeu.

Pour ce jeu, ils ont mieux programmer l'interface graphique, mais les méthodes pour la machine sont un peu faible.

2)

Le programme de Christophe Deprez. Il a mis ce jeu jouer sur ligne: http://christophe.deprez.free.fr/quarto/

Ce jeu est programmé au 6 niveaux. Le niveau 6 est le plus difficile.

II.Technique de la programmation de jeu

II.1 Technique classique

Après le succès de *Deep Blue* en 1997, la technique de la programmation d'échecs est considérée comme modèle pour étudier. Donc des autre jeux de réflexion sont naturellement appliqués par cette technique, qui se compose 3 modules principaux: la Fonction d'Evaluation (***FE***), la Recherche Arborescente (***RA***) et la Génération de Coups (***GC***). Nous appellerons 'achitecture classique ' le triplet (***FE,RA,GC***). Ils sont liés étroitement. ***GC*** aide à générer un arbre , et ***FE*** donne la valeur aux noeuds teminaux, après ***RA*** rend la valeur favorable au noeud racine pour décider quel coup à jouer.

- ***FE***. La fonction d'évaluation est utilisée pour évaluer des positions du jeu. L'évaluation peut être simple (gagnant, perdant, nulle) , ou être complex, soit préciser bien un nombre pour toutes les positions.
- ***GC***. Ce module est pour créer des positions filles à partir d'une postion donnée. Pour programmer ce module, il faut bien connaître les règles de jeu.
- ***RA***. Ce module fonctionne avec *FE* et *GC*. Sur une postion, il appelle *GC* pour créer des positions filles. Sur une postion terminale, il appelle FE pour obtenir la valeur. Sur une position non-terminale, il s'appelle lui-même de manière récursive. Dans la suite, je vais vous présenter deux algorithms de RA: *MinMax* et *AlphaBeta*.

II.1.1)MinMax.

C'est un algorithme qui est posé par John Von Neumann en 1928. Après, à la fin des années 40, ce sont Turing et Shannon qui, les premiers, ont envisagé ce mécanisme de recherche pour l'ordinateurs.

L'idée de cet algorithm est de développer complètement l'arbre de jeu , de noter chaque feuille avec sa valeur, puis de faire remonter ces valeurs avec l'hypothèse que chaque joueur choisit le meilleur coup pour lui. S'il y a un jeu pour 2 joueurs: *J1, J2*. On peut définir la suivante:

- une position gagnante pour *J1* vaut +1
- une position perdante pour *J1* vaut -1
- une position nulle vaut 0

Donc naturellement pour jouer *J1* veut choisir le coup amenant à l'état de plus grande valeur , et *J2* veut choisir le coup amenant à l'état de plus petite valeur . (voir Figure II.1)

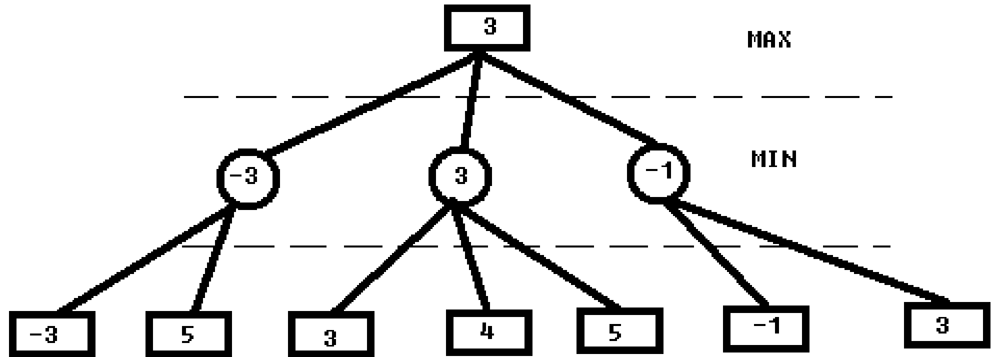

Figure II.1

Dans tous les cas, la valeur d'un noeud est simplement le maximum des opposés de la valeur des fils. L'écriture de la fonction récursive s'en voit simplifiée. On parle de convention *NegaMax*.
En pratique, l'algorithme *MinMax* n'est pas utilsé en l'état, car on constate que l'algorithme minmax doit complètement décrire l'arbre pour fournir une solution, ce qui, on le verra, est souvent excessif.

II.1.2) AlphaBeta.

On observe que l'algorithme *MinMax* effectue l'évaluation pour tous les noeuds de l'arbre de jeu d'un horizon donné. Mais il existe des situations dans lesquelles, pour déterminer la valeur *MinMax* associée à la racine, il n'est pas nécessaire de calculer les valeurs associées à tous les noeuds de l'arbre , et il permet d'optimiser grandement l'algorithme *MinMax* sans en modifier le résultat.
Basé sur *MinMax*, l'algorithme AlphaBeta, qui a été explicitement décrit pour la première fois par Hart et Edwards en 1961, est une amélioration qui réalise un élagage de certaines branches qu'il est inutile de visiter.
La figure II.2 présente les deux types de coupures possibles. Les nœuds Min sont représentés par un rond bleu et les nœuds Max par un carré gris.

Coupure Alpha: le premier fils du nœud Min V vaut 4 donc V vaudra au plus 4. Le nœud Max U prendra donc la valeur 5 (maximum entre 5 et une valeur inférieure ou égale à 4).

Coupure Beta: le premier fils du nœud Max V vaut 4 donc V vaudra au minimum 4. Le nœud Min U prendra donc la valeur 3 (minimum entre 3 et une valeur supérieure ou égale à 4).

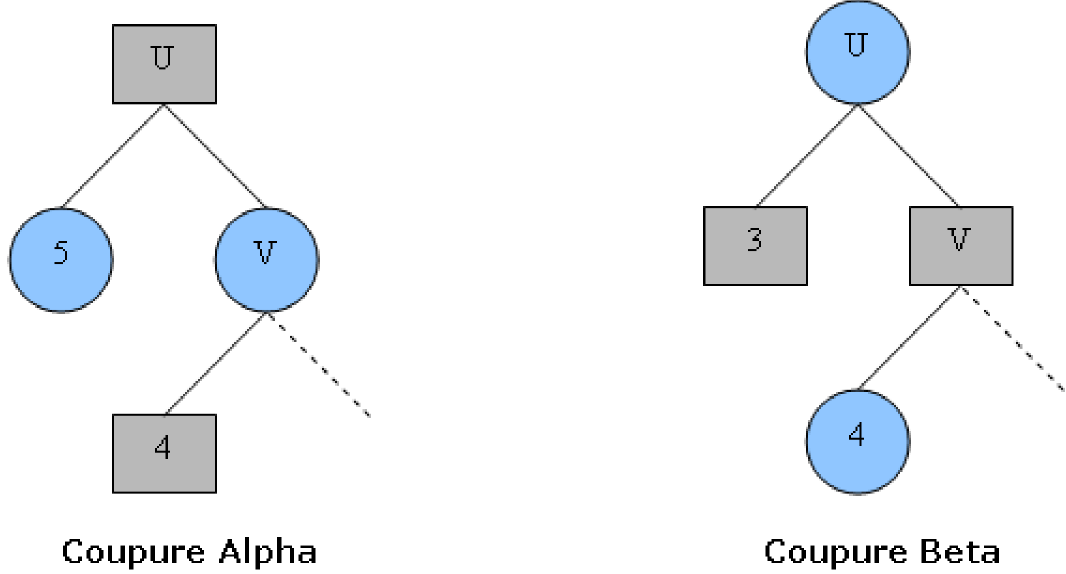

Figure II.2

II.1.3) Autre Améliorations.

- **Itérative Deepenning**

Cette idée (ou stratégie) simple est posé dans les années 70s. En effet, le problème des algorithmes précédents était d'évaluer le jeu à une profondeur fixée. Pour la plupart des jeux, cette recherche est plus ou moins longue (en temps). Si l'on effectue une recherche à une profondeur constante, fixée pour l'ensemble de la partie, cette recherche sera longue en début de partie et beaucoup plus rapide vers la fin. Tout simplement parce que la taille de l'arbre se réduit fortement lorsque le jeu évolue.

Le principe de l'Iterative Deepening est de mettre en place une horloge et lorsque l'on a terminé l'analyse à une profondeur donnée, on évalue le temps qu'il faudra pour une itération supplementaire (une profondeur de plus). Si le temps dépasse celui que l'on s'est fixé, on arrète et on donne le meilleur résultat obtenu. Si par contre, il reste du temps suffisant pour l'ordinateur, on continue la recherche en augmentant la profondeur d'analyse. C'est ainsi que fonctionnent la plupart des programmes qui sont paramétrés en temps, et sont beaucoup plus agréables pour l'utilisateur.

Donc il serait possible d'implanter deux techniques(par exemple: AlphaBeta et Iterative Deepening) pour que l'ordinateur ne consomme pas trop de temps sur un coup pendant jouer l'ensemble de la partie.

Un autre caractère de l'Iterative Deepening est de pouvoir récupérer les informations (les meilleurs coups) issues d'une recherche effectuée à une profondeur inférieure . On profite donc de ces informations de la profondeur n pour classer les coups et faire examiner en premier au programme les meilleurs

coups de la profondeur n lorsqu'il effectue la recherche en profondeur n+1. Parce qu'on sait qu'un algorithme AlphaBeta est d'autant plus efficace qu'il évalue en premier les meilleurs coups.

- **Table de transposition**.

Le principe de la table de transposition est le suivant: pour chaque position rencontrée dans la recherche, l'algorithme retourne une évaluation. Il s'agit de conserver dans une table la valeur de chacune des positions rencontrée de façon à pouvoir réutiliser directement cette valeur lors des recherches suivantes.

Si les tables de transposition fonctionnaient suivant ce principe, un problème se poserait rapidement: comment stocker tous les éléments de la table de transpostion (et en particulier la position de chacune des pièces sur le tablier) dans la mémoire du calculateur, et comment retrouver rapidement une position. On utilise une structure de données connue sous le nom de table de hachage; au lieu d'utiliser la position en entier pour retrouver la valeur de cette position, on utilise un nombre qui caractérise cette position. Et pour générer ce nombre, la technique de Zobrist est plus rapide. Elle utilise l'opérateur XOR pour calculer le nombre(valeur) de chaque position.

Un des problèmes des tables de hachage est que rien ne nous garantit que deux positions différentes ne vont pas avoir la même valeur de hachage, ce qui riquerait de nous amener à des conclusions fausses sur la valeur d'une position. Cette probabilité de collision est d'autant plus faible que la taille de la table est importante. Normalement si l'on prend 64 bits comme la valeur de hachage, la probabilité de collision est négligeable.

Autrement, pour stocker un élément de la table de hachage, on ne peut utiliser la valeur de hachage comme index de tableau car elle est généralement trop grande (un tableau avec un index sur 32 bits occuperait quatre milliards de positions mémoire, ce qui est déraisonnable). On peut par exemple utiliser les 16 premiers bits (ou 20 premiers bits) de la valeur de hachage comme index du tableau. On prend alors le risque que des positions aient des valeurs de hachage différentes et le même index. Il y a alors conflit. Il existe différentes méthodes pour gérer ce type de problème, la plus simple étant de remplacer la plus ancienne des deux valeurs par la plus récente.

La figure II.3 présente comment stocker l'information d'une position dans le table de hashage.

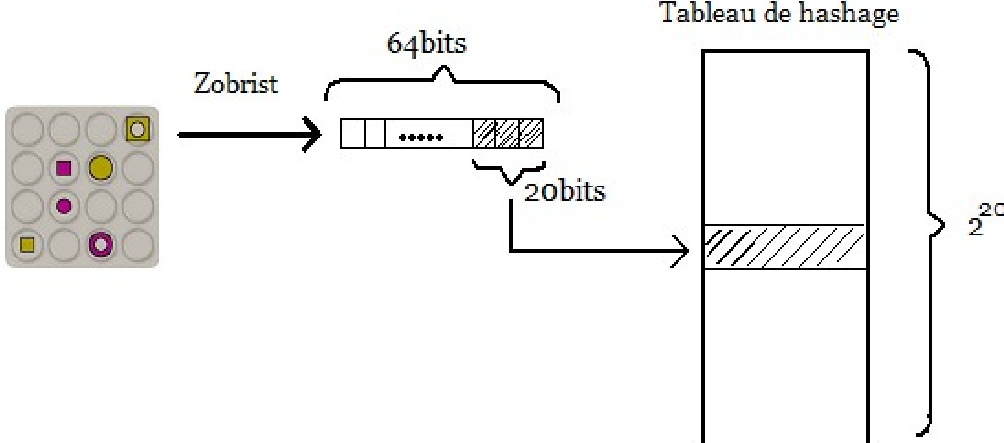

Figure II.3

II.2 Technique Monte-Carlo

Parmi les jeux de réflexion, Go est un jeu exceptionnel. Il est difficile de le réaliser en haut niveau par la technique classique. Donc des chercheurs ont re-étudié la méthode Monte-Carlo sur le GO pendant ces années.

La méthode Monte-Carlo est une technique probabilistes fondée sur la simulation informatique de variables aléatoires. On utilise cette méthode pour réaliser la recherche arborescente. L'idée est : Dans l'arbre qu'on a déjà construit, à partir d'une feuille, ce jeu est simulé jusqu'à la fin de ce partie. Cette simulation est exécutée itérativement, et chaque fois, un noeud qui mène plus de fois gagnante est choisi, en même temps l'arbre est reconstruit quand un noeud nouveau est choisi.

Cette technique peut être défini comme quadruple:(Sélection, Expansion ,Simulation et Retour).

1. Sélection: C'est une fonction qui, pour un noeud non-terminal, choisit un de ses noeuds fils. Elle contrôle la balance entre l'exploitation et l'exploration. C'est-à-dire, cette fonction peut choisir un noeud qui mène le meilleur résultat de la simulation, et aussi peut choisir un noeud qui n'est pas exploré. Ca correspond au problème de MAB(Multi-Armed Bandit).
   Mais comment faire? On utlise l'algorithme *UCT*: Upper Confidence bound applied to Trees(Kocsis and Szepesvri,2006). La fonction chosit un noeud i qui maximise:

$$V_i + c\sqrt{\frac{\ln N}{N_i}}$$

   où Ni est le nombre de jeux simulés à partir du noeud i, et N est le nombre

total de jeux simulés à partir du père du noeud i. Vi est l'évaluation de ce noeud i, soit le moyen des résultats des simulations. Par example, pour un noeud, 3 fois simulé, les résultats sont 1(gagnant), -1(perdante) et 0(nulle). Donc Vi = (1-1+0)/3 =0.

2. Expansion: C'est un stratégie pour stocker un noeud dans la mémoire. Pendant des simulations, l'arbre est développé progressivement. On peut utiliser la table de transposition pour stocker l'arbre qui est exploré.

3. Simulation. C'est une fonction qui, à partir d'un noeud(position de jeu), simule le jeu jusqu'à la fin de la partie. Pendant le processus, des coups à jouer sont choisis aléatoirement.

4. Retour. C'est une fonction qui retourne et modifie les valeurs de noeuds de l'arbre selon des résultats des simulations précédées.

La figure II.4 présente le schéma de cette technique.

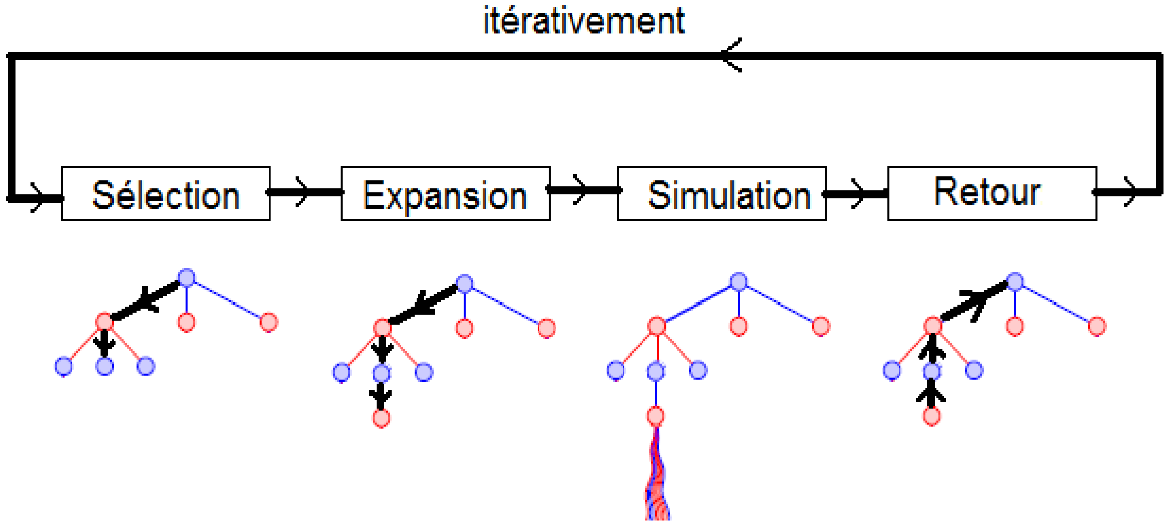

Figure II.4

Il y a 2 points différents entre la technique classique et la technique Monte-Carlo:

- La recherche d'arborescente: Dans la technique classique, la recherche est déterministe. Tous les noeuds possibles sont crées. Parfois, la recherche ne peut pas atteindre des noeuds terminaux à cause du temps limite, du matérielle, et de la factor de branchement. Mais dans la technique Monte-Carlo, la recherche est aléatoire, l'arbre n'est pas complètement construit, mais il est développé progressivement.
- Evaluation. On peut définir la fonction de l'évaluation plus complex dans la techniques classique pour ces noeuds non-terminaux; Et la technique Monte-Carlo n'a pas besoin d'une évaluation complex, juste une évaluation simple (gagnante, perdante, nulle) est suffissante.

III Réalisation de Programme

III.1 L'objectf et la conception

L'objectif de mon stage est d'analyser les différentes méthodes appliquées sur Quarto. Il faut que je puisse vérifier les résultats en détail. Donc j'ai décidé de programmer en C++, sous le système Linux. Les résultats sont affichés dans la fenêtre terminal.

Quant à la conception de ce programme, j'espère que mon programme peut réaliser les fonctions suivantes:

1.Jouer à 2 personnes. C'est juste de suivre la règle de Quarto pour jouer.

2.Jouer à un joueur contre la machine. On peut choisir une méthode pour la machine par régler des paramètres.

3.Jouer à 2 machines. Le programme est lancé sur un ordinateur, mais il peut simuler le jeu entre 2 machines. D'abord je choisis les méthodes et les paramètres respectivement pour deux machines. Après je peux définir combien de parties à jouer . Les parties de jeu peuvent être lancées automatiquement, et à la fin les résultats (le nombre de partie gagnante pour deux machines respectivement,le nombre de partie nulle) vont être retournés .

III.2 Les objets et l'architecture

Pour réaliser ce jeu Quarto en appliquant les techniques, j'ai défini 4 modules dans mon programme: Tablier, FE(Fonction d'Evaluation), GC(Génération de Coups) et RA (Recherche Arborescente). Ils se composent respectivement d'un ou de plusieurs objets (classes) au niveau de la programmation.

1.**Tablier**: Ce module est de réaliser toutes les opérations sur le tablier comme: initialiser le tablier, afficher le tablier dans le terminal, actualiser le tablier après jouer un coup, et aussi justifier si la partie du jeu est fini ou pas. Ce module se compose 2 classes (Damier et CheckGameOver).

2.**FE**. Ce module est de évaluer la situation du tablier et de retourner une valeur. Il correspond à la fonction d'évaluation. Ce module n'a qu'une classe (Evaluation).

3.**GC**. Ce module est utilisé pour générer tous les coups possible à partir d'une position dans le tablier. Il correspond à la Génération de Coup. Pour Quarto, en raison de sa règle particulière, ici un coup à jouer est composé par deux parties: choisir une position et choisir un pion. Par exemple, si dans le tablier, il reste encore 5 positions à déposer et 5 pions à choisir, donc à partir de cette

position, on a 25 (5x5) coups possibles à jouer. Ce module a une seul classe (GenerateMove).

4.**RA**. C'est le module de la Recherche Arborescente. Il est en charge de trouver le meilleur coup à jouer selon sa capacité. Ce module se composent des classes qui correspondent aux méthodes dont j'ai parlé: *MinMax*, AlphaBeta, Itérative Deepenning, Table de Transposition, *UCT*, et des méthodes mélangées.

L'architecture des modules et des classes peut être décri comme la figure suivante(Figure III.1):

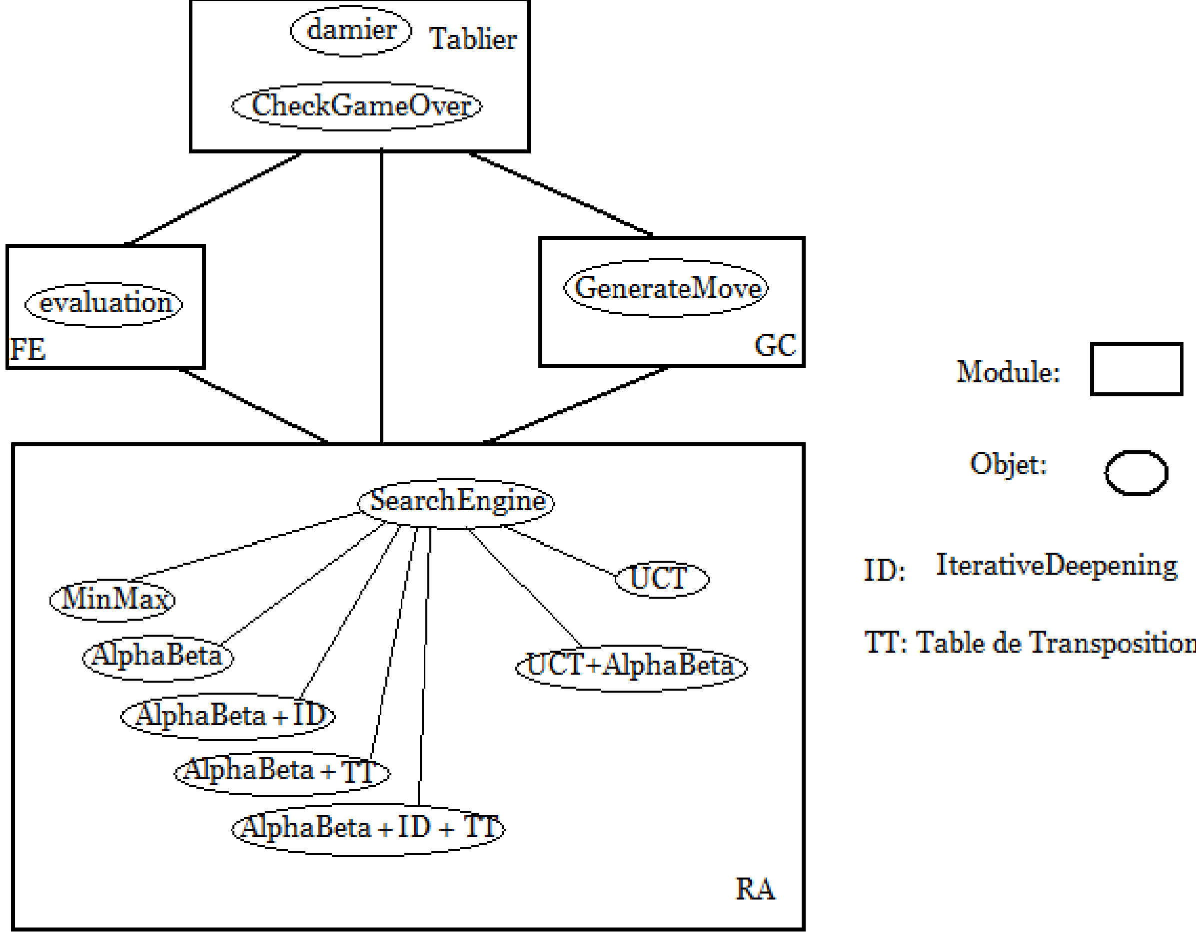

Figure III.1

III.3 La programmation

Dans cette partie, je vais présenter la programmation pour les classes dont j'ai parlé ci-dessus.

1).Damier:

C'est une classe qui peut afficher le tablier dans le terminal, initialiser le tablier pour commencer une partie de jeu, et actualiser le tablier après jouer un coup. (Le programme se trouve sur Damier.cpp, voir Annex.)

Comment définir le tablier de Quarto et l'afficher dans la fenêtre terminal? C'est le premier problème à résoudre.

On sais que chaque pion a 4 caractère (bois clair ou bois foncé, petit ou grand, plein ou troué, paralléppipèdique ou cylindrique) , en total il y a 8 caractères. Je vais les transformer en nombre:

Caractère	nombre	Caractère	nombre
bois clair	**1**	bois foncé	**2**
cylindrique	**3**	paralléppipèdique	**4**
grand	**5**	petit	**6**
plein	**7**	troué	**8**

Par exemple, pour un pion (bois clair, cylindrique,petit, toué), on peut le présenter par 1368.

Donc, les 16 pions peuvent décrit comme la suivante:

Pion1 (**1468**)		Pion2 (**1467**)	
Pion3 (**1458**)		Pion4 (**1457**)	
Pion5 (**1368**)		Pion6 (**1367**)	

Pion7(**1358**)		Pion8 (**1357**)	
Pion9 (**2468**)		Pion10 (**2467**)	
Pion11 (**2458**)		Pion12 (**2457**)	
Pion13 (**2368**)		Pion14 (**2367**)	
Pion15 (**2358**)		Pion16 (**2357**)	

Le pion est défini comme objet dont la structure est:

```
typedef struct _pions
{
   POSITION  position;  // La position actuelle du pion
   char *caractere;     //les 4 caractères du pion
}PION;
```

Et le damier initialisé est affiché comme la figure III.2:

```
     1     2     3     4   [j]
   -------------------------
 1 |     |     |     |     |
   -------------------------
 2 |     |     |     |     |
   -------------------------
 3 |     |     |     |     |
   -------------------------
 4 |     |     |     |     |
   -------------------------
[i]

 pion[1]=1468  pion[2]=1467  pion[3]=1458  pion[4]=1457
 pion[5]=1368  pion[6]=1367  pion[7]=1358  pion[8]=1357
 pion[9]=2468  pion[10]=2467  pion[11]=2458  pion[12]=2457
 pion[13]=2368  pion[14]=2367  pion[15]=2358  pion[16]=2357
```

Figure III.2

2).CheckGameOver.

C'est une classe pour justifier la fin de partie du jeu. On sais que , pour Quarto, la condition de victoire a quatre niveaux. Dans ce programme, on réalise juste le niveau 1. C'est-à-dire si quatre pions qui ont le même caractère se trouvent sur même rang aligné (ligne, colonne, diagnale), le joueur va gagner. (Voir la figure III.3) Le programme se trouve sur CheckGameOver.cpp.(Voir Annex)

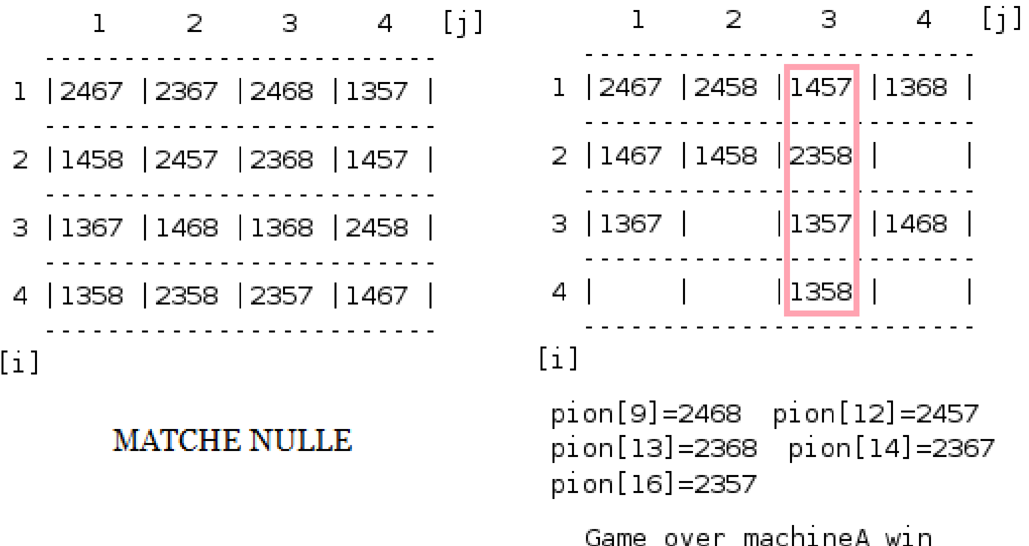

Figure III.2

3).Evaluation.

Cette classe est utilisé pour évaluer la valeur de la position dans le tablier, soit les noeuds dans l'arbre de la recherche. Elle est appellé par RA récursivement pendant la recherche arborescente.

La fonction d'évaluation est un factor crucial pour la technique classique. Une évaluation simple est de seulement évaluer le résultat du jeu (gagnant, perdant, null). Et une évaluation complex est de pouvoir évaluer les positions par une valeur assez précise. Dans la technique classique, un jeu avec la fonction d'évaluation complex et correcte est meilleur que celui avec la fonction d'évaluation simple. Mais le problème pour l'évaluation complex est comment définir la valeur précise et correct pour une position non-terminal. Donc il faut que le programmeur ait une bonne connaissance sur le stratégie de gagner ce jeu . Un autre problème est que l'évaluation complex va prendre plus de temps pour évaluer que l'évaluation simple.

Pour la technique Monte-Carlo, elle n'a pas besoin d'une évaluation complex parce qu'elle peut faire la recherche arborescente jusqu'à la fin du partie du jeu.

Donc, j'ai choisi l'évaluation simple à programmer pour Quarto. Pour deux joueurs (machine et humain), il faut définir les différentes valeurs pour appliquer la technique classique. Comme la figure suivante:

	Gagnante	**Perdante**	**Autre cas**
Machine	**-9999**	**9999**	**0**
Humain	**9999**	**-9999**	**0**

Pour la technique Monte-Carlo, la définition est plus simple. On n'a besoin de définir que la valeur pour la machine.

	Gagnante	**Perdante**	**Nulle**
Machine	**1**	**-1**	**0**

4).GenerateMove

C'est une classe pour génerer tous les coups possibles à partir d'une position. Elle est appellée récursivement par RA pour construire l'arbre de la recherche. Les coups sont enregistrés dans un tableau à deux dimensions (movelist [layer] [movenumber]) où layer est la profondeur de la recherche.

Quant à la structure du coup , elle est définit comme un objet:

```
typedef struct _pionmove
{
int  PreScore;//Le pion qui va être déplacé
POSITION  to; //La position où joueur déplace le pion 'prescore'
int  Score;    //Le pion que joueur choisit après
```

```
}MOVE;
```

5).SearchEngine

La classe SearchEngine est une classe père qui est hérité par les 8 classes dans le module RA: *StochaEngine, MinMaxEngine, AlphaBetaEngine, IDAlphaBetaEngine, TTAlphaBetaEngine, IDTTAlphaBetaEngine, UCTEngine et AlphaBetaEngineUCT.* Chaque classe représente une méthode différente.
Quant à la classe SearcheEngine, elle réalise des méthodes communes:

- MakeMove: jouer un coup sur un tablier copié.
- UnMakeMove: annuler le coup précédent qui est joué sur le tablier copié.
- IsGameOver: déterminer si un noeud terminal, soit la fin de la partie du jeu simulé par la machine, est arrivé .

Une méthode SearchGoodMove, qui présente la technique appliqué sur la machine, est laissée compléter par les 8 classes dont j'ai parlé.

6)StochaEngine

Cette classe réalise simplement une machine qui joue ce jeu par choisir le coup aléatoirement. Evidemment, cette classe n'a pas besoin de paramètre pour régler.

7)MinMaxEngine

Cette classe réalise d'utiliser l'algorithme *MinMax* pour faire la recherche arborescente. Quant à la programmation, j'utilise NegaMax pour simplifier cet algorithme. Le pseudocode est comme la suivante:

```
MinMaxEngine::SearchGoodMove(damier,depth,BestList)
{
CurrentDamier=damier; //copy le damier réel
SearchDepth=depth;    // Donner la profondeur à chercher
MinMax(SearchDepth);   //Utiliser l'algorithme MinMax
return bestmove;//Après la recherche,retourner le meilleur coup
};
MinMaxEngine::MinMax(depth)        // l'algorithme MinMax
{//initialiser une valeur assez petite.
 best=-infini;
//Si c'est un noeud terminal, retourner le résultat du jeu
 if(IsGameOver(CurrentDamier,depth))   return résultat ;
//Si c'est un noeud feuille, retourner la valeur évaluée
```

```
if(depth<=0)
    return Evaluate(CurrentDamier,(SearchDepth+1-depth)%2);
//créer tous les coups possible pour le damier actuel
 count = CreatePossibleMove(CurrentDamier, depth);
 for(i=0;i<count;i++)
 {//jouer un coup
  MakeMove(movelist[depth][i]);
 //Apeller l'algorithme MinMax récursivement
  val=-MinMax(depth-1);
  //annuler le coup précédent
  UnMakeMove(movelist[depth][i]);
   if(val>best)
   { //enregistrer la meilleur valeur
     best=val;
     // enregistrer le meilleur coup pour le noeud racine
     if(depth==MaxDepth)    bestmove=movelist[depth][i];
   }
return best;
  }
}
```

Pour utiliser MinMaxEngine, on définit 2 paramètres: la profondeur (D) où D>=1 et BestList (B) où B=0/1. Ll'objectif de l'utilisation de BestList est de enregistrer les coups de la même valeur comme le meilleur coup, je vais en parler en détail à III.4

8)AlphaBetaEngine

Cette classe réalise l'algorithme *AlphaBeta* pour faire la recherche arborescente. De la même manière que l'algorithme *MinMax* peut être remplacé par NegaMax, on simplifie Alpha-Beta. Le pseudocode est la suivante:

```
AlphaBetaEngine::SearchGoodMove(damier,depth,ENumber,BestList)
{
 CurrentDamier=damier; //copy le damier réel
 SearchDepth=depth;// Donner la profondeur à chercher

//Au début de partie, utiliser la façon stochastique pour jouer
if(EtatValide(CurrentDamier))
          Stochastique.SearchGoodMove(CurrentDamier);
else
     {AlphaBeta(SearchDepth,-99999,99999);//l'algo AlphaBeta
```

```
       return bestmove;
      }
};

AlphaBetaEngine::AlphaBeta( depth, alpha, beta)
{
//Si c'est un noeud terminal, retourner le résultat du jeu
 if(IsGameOver(CurrentDamier,depth))   return résultat ;
//Si c'est un noeud feuille, retourner la valeur évaluée
if(depth<=0)
    return Evaluate(CurrentDamier,(SearchDepth+1-depth)%2);
//créer tous les coups possible pour le damier actuel
 count = CreatePossibleMove(CurrentDamier, depth);

 for(i=0;i<count;i++)
 {//jouer un coup
  MakeMove(movelist[depth][i]);
  //Apeller l'algorithme MinMax récursivement
  val=-AlphaBeta(depth-1,-beta,-alpha);
  //annuler le coup précédent
  UnMakeMove(movelist[depth][i]);

if (val> alpha)
       {alpha = val;
       if(depth == MaxDepth) bestmove=movelist[depth][i];
         }
if (val >= beta)    break;
 }
return alpha;
}
```

Pour utiliser AlphaBetaEngine, on définit la valeur de la profondeur (D) où D>=1. Au début de programme, j'utilise le StochaEngine aussi. C'est une amélioration et je vais en parler en détail à III.4)

9)IDAlphaBetaEngine

Cette classe réalise l'algorithme *AlphaBeta* en appliquant la méthode *Itérative Deepenning* .

```
IDAlphaBetaEngine::SearchGoodMove( damier, time, ENumber)
{
```

```
CurrentDamier=damier;
timeID=time; //temps de réflechir

for (MaxDepth = 1; MaxDepth <= 16; MaxDepth++)
    {
        if (has time)
        IDAlphaBeta(MaxDepth,-99999,99999);
        else break;
    }
return bestmove;
};

int  IDAlphaBetaEngine::IDAlphaBeta(int  depth,int  alpha,int
beta)
{
// Ici Comme AlphaBeta, en plus la contrôle du temps
}
```

Pour l'*Itérative Deepenning*, le paramètre à définir est le temps (*T*) de réflexion pour la machine . Ici, pour utilisateur, le temps est donné pour toute la partie. On définit le temps *T* pour jouer une partie et un constant *C* (ici *C*=10), le temps *T i* pour jouer le coup i :

$$Ti = \frac{T - \sum_{i=1}^{i-1} Ti}{C}$$

Par exemple, l'utilisateur donne 30 seconds pour cette méthode. Donc la machine va utiliser 3 seconds à réfléchir pour le premier coup, et le deuxième coup a (30-3)/10 =2.7 seconds à réflechir.

10)TTAlphaBetaEngine

Cette classe réalise l'algorithme *AlphaBeta* en appliquant la *Table de Transposition* . Pendant la recherche arborescente, avant explorer un noeud , cette méthode va lire la *able de Transposition* pour voir s'il existe. S'il existe, l'information de sa valeur va être récupéré. Sinon, cette méthode va faire la recherche d'AlphaBeta normale pour trouver la valeur, et à la fin enregistrer ce noeud et sa valeur dans la *able de Transposition*.

La table de transposition est définit comme un objet qui réalise de génerer HashKey de 64 bits et 32 bits; calculer le nombre de zombrist d'une position;

enregistrer les informations du noeud et chercher les informations du noeud. Les détails sont dans TableTransposition.cpp en Annex.

Pour l'utilisateur, cette méthode a beoin d'un paramètre à définir: la profondeur de la recherche (*D*) , comme la méthode *AlphaBeta.*

```
TTAlphaBetaEngine::SearchGoodMove(damier,depth)
{CurrentDamier=damier;
//Calculer le HashKey de la position racine
CalculateInitHashKey(CurrentDamier);
SearchDepth=depth;

if(EtatValide(CurrentDamier))
     Stochastique.SearchGoodMove(CurrentDamier);
else
{TTAlphaBeta(SearchDepth,-99999,99999);
 return bestmove;
}
};

TTAlphaBetaEngine::TTAlphaBeta( depth,alpha, beta)
{
 if(IsGameOver(CurrentDamier,depth))  return over;
//Chercher ce position dans la table pour voir s'il existe.
LookUpHashTable(alpha,beta,depth,(SearchDepth+1-depth)%2);
//S'il existe, retourner la valeur du noeud directement
if(exister) {return valeur;}
//Sinon, faire la recherche AlphaBeta
 if(depth<=0)
    return Evaluate(CurrentDamier,(SearchDepth+1-depth)%2);
 count = CreatePossibleMove(CurrentDamier, depth);
 for(i=0;i<count;i++)
 {//Méthode de TT, pour générer le HashKey quand jouer un coup
  Hash_MakeMove(movelist[depth][i]);
  MakeMove(movelist[depth][i]);
  score=-TTAlphaBeta(depth-1,-beta,-alpha);
  //Méthode de TT, pour récupérer le HashKey précédent
  Hash_UnMakeMove(movelist[depth][i]);
  UnMakeMove(movelist[depth][i]);
   if (score >= beta)
   {//Enregistrer la valeur à TT
    EnterHashTable(score,depth,(SearchDepth+1-depth)%2);
   return score;
```

```
    }
   if (score > alpha)
   {  alpha = score;
      if(depth == MaxDepth)  bestmove=movelist[depth][i];
   }
 }
////Enregistrer la valeur à TT
   EnterHashTable(alpha,depth,(SearchDepth+1--depth)%2);
return alpha;
```

11)IDTTAlphaBetaEngine

Cette classe réalise l'algorithme *AlphaBeta* en appliquant *Itérative Deepenning* et la *Table de Transposition*. Pour l'utilisateur, on n'a besoin de définir qu'un seul paramètre : le temps de réflexion (*T*) comme la méthode *Itérative Deepenning*.

```
IDTTAlphaBetaEngine::SearchGoodMove(damier,time)
{CurrentDamier=damier;
 CalculateInitHashKey(CurrentDamier);
 timeID=time;
for (MaxDepth = 1; MaxDepth <= 16; MaxDepth++)
   {if (has time)
      IDAlphaBeta(MaxDepth,-99999,99999);
      else break;
   }
return bestmove;
};

IDTTAlphaBetaEngine::IDTTAlphaBeta(depth, alpha, beta)
{
//ici comme TTAlphaBeta,, en plus la contrôle du temps
}
```

12)UCTEngine

Cette classe réalise la technique Monte-Carlo dont la sélection du coup à jouer est appliquée l'algorithme *UCT*. La classe *UCT* Engine se compose de 3 méthodes principaux:

- SearchGoodMove : C'est la recherche arborescente pour retourner le

meilleur coup.

- SelectMove: Méthode de Sélection par l'algorithme *UCT*
- PlaySimulateGame: Cette méthode réalise un jeu simulé à partir d'un noeud non-terminal , et retourner le résutlat.

En outre, elle a besoin d'une autre classe *UCTTT* pour enregistrer l'arbre qu'elle construit dans la table de transposition (méthode EnterHashTable()), et pour modifier la valeur du noeud après la simulation (méthode UpdateValue()). Les détails de *UCTTT* se trouve à UCTTT.cpp en Annex.

Quant aux paramètres pour cette classe, il faut définir le temps de réflexion (T) et le constant (C) de *UCT*.

```
UCTEngine::SearchGoodMove(damier,T,C)
{
while( has time T) do
{   //La phase Sélection
    while( HashKey du noeud existe à TT) do
      {     //Méthode de Sélection
            SelectMove();
            //Générer le HashKey
            Hash_MakeMove();
            //Jouer le coup choisi
            MakeMove;
       }

     //La phase Expansion:enregister le noeud à TT
     EnterHashTable();

     //La phase Simulation
     //Méthode de Simuler
     PlaySimulateGame();
      //Résultats de la simulation
      if(machine gagne) result=1;
      if(machine perd) result=-1;
      if(match null) result=0;

    //La phase Retour
    while(ce noeud existe à TT et  !racine) do
    {    //Modifier les valeurs des noeuds à TT
         UpdateValue(result);
         //récupérer le Hashkey précédent
         Hash_UnMakeMove();
```

```
        //annuler le coup précédent
        UnMakeMove;
    }
}
return BestMove;
}

//Méthode de sélection
UCTEngine::SelectMove(damier,depth)
{
count = CreatePossibleMove(CurrentDamier);
for(int i=0;i<count;i++)
 {
//Si ce coup n'est jamais joué, donner lui une grande valeur.
  if(NodeNb[i]==0) NodeValue[i]=10000;
  else
//Sinon,utiliser UCT pour calculer la valeur
  NodeValue[i]=NodeValue[i]/NodeNb[i]+sqrt(C*log(nb)/NodeNb[i]);
 }
return NodeMaxValue;
}

//Méthode de simulation
UCTEngine::PlaySimulateGame( damier, depth)
{
//Si un noeud terminal est arrivé, retourner le résultat
if(IsGameOver)  return over;

//Sinon, faire la simulation
do
{
// choisir un coup aléatoirement pour jouer
count = c.CreatePossibleMove(NDamier, 1);
time_t grain;
srand(time(&grain));
int i;
i=rand()%count;
move=movelist[1][i];

//Jouer ce coup dans un damier copié
NDamier.CurrentDamier(move.to.x,move.to.y,move.PreScore);
```

```
NDamier.NPionsAJouer=move.Score;

}while(!IsGameOver);

//retourner le résultat
return over;
}
```

13)AlphaBetaEngineUCT

Cette classe réalise une méthode mélangée de l'algorithme AlphaBeta et l'algorithme *UCT*. L'idée est la suivante: Au départ de la partie du jeu, on utilse la méthode UCT, après dans la dernière phase du jeu on utilise la méthode AlphaBeta. Il faut définir 4 paramètres pour l'utilisateur: le temps de reflexion *(T)* , le constant *(C)* comme la méthode *UCT*; la prfondeur de la recherche *(D)* pour la méthode AlphaBeta, en plus, un nombre *(N)* qui décide quand on change la méthode *UCT* à celle de l'AlphaBeta. Par exemple, si *N*=9, c'est-à-dire quand le nombre de pions qui ne sont pas joués est supérieur à 9, on va utiliser *UCT*. Sinon, on va utiliser l'AlphaBeta.

```
MOVE AlphaBetaEngineUCT::SearchGoodMove(damier,D, N,T, C)
{
 CurrentDamier=damier;
//Si le nombre des pions restes >=N, utiliser UCT
if(EtatValide(CurrentDamier,N))
UCTEngine.SearchGoodMove(CurrentDamier,T,C);
//Sinon, utiliser AlphaBetaEngine de profondeur D
else
AlphaBetaEngine.SearchGoodMove(CurrentDamier,D);
};
```

III.4) Amélioration de la programmation

Pour obtenir les meilleurs résultats, et pour faire une annalyse correcte, il existe les façons empiriques pour améliorer le programme.

- **Utiliser la méthode stochastique**

Au début de la partie de Quarto, évidemment le choix du premier pion et le choix pour où le déposer ne vont pas influencer le résultat du jeu. Donc, avant commencer une partie, le premier pion peut être choisi aléatoirement.

Après, peut-être on peut se demander, pour Quarto, quelle condition est negligable pour le résultat du la partie du jeu? Selon ma connaissance du jeu Quarto, quand il n'y a que un ou deux pions sur le même rang aligné(ligne, colonne, diagonale), le résultat du jeu ne va pas être influencé à partir de cette situation. Ca ressemble au problème de N-Queens. Ici $N=4$, donc au maximum on peut jouer 5 pions aléatoirement au début. Par exemple, pour la position dans la Figure III.3, le coup prochain peut être encore choist aléatoirement.

Donc, cette amélioration est appliquée dans les classes AlphaBetaEngine et TTAlphaBetaEngine, Pour la machine, elle n'a pas besoin de prendre beaucoup de temps pour donner le premier ou deuxième coup. Et ça donne plus de la partie variable.

En outre, pour bien utiliser la méthode stochastique, c'est mieux d'utliser la fonction srand(graine) pour obtenir plus des parties variable.

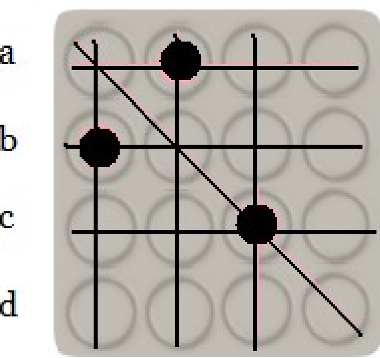

Figure III.3

- **Utiliser HashKey**

Quand on lance 100 parties du jeu entre 2 machine de différente méthode , est-ce que c'est possible d'avoir 2 ou plus des parties pareilles? Oui, c'est possible.

Pour analyser les résultats plus correctement, il faut enlever les parties pareilles. Mon idée de juger si deux parties sont pareilles est simple: juger le nombre H . Pour une partie i

$$Hi=Ni + Mi.$$

Où Ni est l'index du premier pion

Mi est le hashkey du tablier terminal

J'ai utilisé le hashkey de la table de transposition parce que pour chaque poistion différente du tablier, le hashkey est différent aussi.
Donc, pour deux partie *j* et *k* si *Hj* =\= *Hk*, c'est sûr qu'ils sont différentes. Si *Hj* = *Hk*, elles ont encore la possibilité d'être différente , mais je ne vais pas le compter dans mon analyse.

- **Utiliser BestList**

On sais que dans la recherche arborescente, la machine va choisir un noeud qui a le meilleur valeur. S'il y a plusieurs noeuds qui ont la même valeur meilleure, la machine va choisir le premier noeud qu'elle a trouvé. BestList est un tableau pour enregistrer ces noeuds. Et la machine va choisir aléatoirement un noeud dans ce tableau.
Cette idée est pour avoir plus de partie variable et elle est appliquée seulement sur la classe *MinMax*. En effet, elle ne peut pas augmenter le niveau de la méthode. On n'a pas besoin de l'appliquer sur les autres classes, parce que on peut déjà avoir des parties différente par l'utilisation de la méthode stochastique dont j'ai parlé.

IV Expérience et résultat

Dans cette partie, je vais présenter les résultats de jeu entre 2 machines (méthodes). L'objectif est de comparer ces différentes méthodes, de voir leur avantages, leur inconvénients, et de trouver la plus forte , c'est-à-dire qu'une méthode gagne plus des jeux. Ma machine d'essai est Fujitsu&Simens Amilo Li1705(CPU core Duo T2060, 768M mémoire).

Il y a 8 méthodes:StochaEngine, *MinMaxEngine, AlphaBetaEngine, IDAlphaBetaEngine, TTAlphaBetaEngine, IDTTAlphaBetaEngine, UCTEngine et AlphaBetaEngineUCT.* Pour bien présenter les résultat, chaque méthode peut être décrit comme :

- StochaEngine: *S,* il n'y a pas de paramètres pour cette méthode.
- MinMaxEngine: *M<D,B>* où *D* est la profondeur pour la recherche, et *B* est le paramètre pour utiliser la fonction BestList. Par exemple, *M<3,0>* signifie que la méthode *MinMax* fait la recherche à la profondeur 3, et n'utilise pas le BestList.
- AlphaBetaEngine: *A<D>* où *D* est la profondeur pour la recherche.
- TTAlphaBetaEngine: *TA<D>* où *D* est la profondeur pour la recherche.
- IDAlphaBetaEngine: *IA<T>* où *T* est le temps pour la recherche.
- IDTTAlphaBetaEngine: *ITA<T>* où *T* est le temps pour la recherche.
- UCT : *U<T,C>* où *T* est le temps pour la recherche, *C* est le paramètre pour l'algorithme UCT.
- AlphaBetaEngineUCT: *UA<D,N,T,C> où D* est la profondeur pour la recherche*, N* est un nombre entre 1 et 16 qui décide quand on change la méthode UCT à celle de l'AlphaBeta*, T* est le temps pour la recherche*, C* est le paramètre pour *UCT.*

Les résultats des parties sont interprétés par 6 aspects:

1)N-gagner: le nombre des fois gagnante, soit combien de fois la méthode gangne.
2)N-null: le nombre des fois du matche null
3)T-gagner: le taux de gagner
4)N-pl: le nombre des parties lancées
5)N-pd: le nombre des parties différentes
6)TIME: le temps moyen pour jouer une partie
7)N-noeud: le nombre des noeuds explorés pour jouer un coup

1)StochaEngine *S*

J'ai lancé 120 parties entre StochaEngine et MinMaxEngine M<2,1>, et aussi 900 parties entre StochaEngine et AlphaBetaEngine A<2>. Les résultats sont décrit dans la table suivante:

	JEU 1		JEU 2	
	S	M<2,1>	S	A<2>
N-gagner	0	120	0	167
N-null	0		0	
T-gagner	0%	100%	0	100%
N-pd / N-pl	120 / 200		167 / 900	
TIME	48.65s/200 =0.24 s		24.53s/900 = 0.03 s	

Table IV.1

Evidemment c'est une méthode plus rapide, mais aussi plus faible que les autre méthodes.

2)MinMaxEngine *M*

Pour MinMaxEngine, j'ai lancé des parties entre MinMaxEngine de différente paramètres, comme la table suivante:

	JEU 3		JEU 4		JEU 5	
	M<1,0>	M<2,0>	M<1,1>	M<2,1>	M<2,1>	M<3,1>
N-gagner	1	8	43	106	4	18
N-null	4		50		8	
T-gagner	7.7%	61.5%	21.6%	53.3%	13.3%	60%
N-pd / N-pl	13 / 100		199/ 300		30 / 30	
TIME	26.39s/100 =0.26 s		77.88s/300 = 0.26 s		1114.63s/30=37.15s	

Table IV.2

Le jeu 3 peut montrer que ,sans utiliser BestList, la plus part des parties sont identiques parmi les 100 parties lancées. Donc pour une bonne analyse, il faut utilser BestList pour *MinMax*.
En théorie, si la méthode fait la recherche de l'arbre plus profonde, son capacité de jouer est plus fort. Donc c'est normale que M<2,1> gagne plus que M<1,1>. Aussi M<3,1> est plus fort que M<2,1>. Mais le problème pour *MinMax* est que M<3,1> prend beaucoup de temps pour jouer un coup au départ du jeu, comme la table IV.3.

	M<2,1>	M<3,1>
Temps pour jouer 1er coup de partie	0.26s	32.45s
N-noeud pour 1er coup de partie	50640	9223440

Table IV.3

3)AlphaBetaEngine *A*

On sais que AlphaBeta est l'amélioration de *MinMax*. Je lance du jeu entre M<2,1> et A<3> pour comparer le résultat du jeu entre M<2,1> et M<3,1>. En outre , je lance des jeux pour voir les différences sur AlphaBetaEngine avec la profondeur différente.

	JEU 6		**JEU 7**		**JEU 8**	
	M<2,1>	**A<3>**	**A<1>**	**A<2>**	**A<2>**	**A<3>**
N-gagner	10	98	47	81	36	56
N-null	42		25		16	
T-gagner	6.7%	65.3%	30.7%	52.9%	33.3%	51.9%
N-pd / N-pl	150 / 150		153 / 800		108 /180	
TIME	86.2s/150 =0.57 s		33.59s/800 = 0.04 s		54.94s/180=0.31s	

	JEU 9		**JEU 10**		**JEU 11**	
	A<3>	**A<4>**	**A<4>**	**A<5>**	**A<5>**	**A<6>**
N-gagner	16	70	45	51	5	6
N-null	34		4		9	
T-gagner	13.3%	58.3%	45%	51%	25%	30%
N-pd / N-pl	120 / 120		100 / 100		20 / 20	
TIME	324.49s/120 =2.7 s		2007.36s/100 = 20.07 s		1713.44s/20=85.67s	

Table IV.4

Par le résultat du jeu 6(M<2,1> Vs A<3>), on peut voir que AlphaBeta beaucoup améliore *MinMax* à TIME en raison de la réduite du nombre des noeuds explorés. Quant aux AlphaBetaEngine de profondeur(D) différente, A<D+1> est plus fort que A<D>. Et quand le D est plus grand, la disparité entre D+1 et D est plus petite, sauf le jeu 9(A<3> Vs A<4>) qui est un peu abnormal. Le jeu 11(A<5> Vs A<6>) prend beaucoup de temps , donc ce n'est pas possible de lancer des jeux avec D plus profond.

	A<3>	**A<4>**	**A<5>**
Temps pour jouer 1er coup de partie	0.35s	5.68s	34.93s
N-noeud pour 1er coup de partie	44637	139154	6860847

Table IV.5

Une méthode raisonnable peut équilibrer le temps de réfléchir et la puissance, donc j'utilise l'Iterative Deepenning et la Table de Transposition.

4)*TA* , *IA* et *ITA*

Quarto est un jeu particulier en raison de sa règle. Et la Table de Transposition ne fonctionne qu'à partir de la profondeur 5. La table suivante montre les informations où TA joue le 3 ème coup (soit il reste 14 pions à choisir et 14 cases vide à déposer).

	TA<5>	**A<5>**	**TA<6>**	**A<6>**
Temps pour jouer 3ème coup de partie	18.51s	21.48s	161.61s	161.29s
N-noeud pour 3ème coup de partie	2292172	2901181	6388974	6466406

Table IV.6

TA<5> reduit 21% du nombre de noeuds exploré par A<5>, et TA<6> ne reduit que 1.2% du nombre de noeuds exploré par A<6>.

Pour IA, le temps de réflechir T est défini à 10 s, 20s, et 30s.

	JEU 12		**JEU 13**		**JEU 14**	
	IA<10s>	**A<2>**	**IA<10s>**	**A<3>**	**IA<20s>**	**A<3>**
N-gagner	73	10	52	14	59	13
N-null	35		52		44	
T-gagner	61.9%	8.5%	44.1%	11.9%	50.9%	11.2%
N-pd / N-pl	118 / 120		118 / 120		116 / 120	
TIME					1101.75s/116=9.5 s	

	JEU 15		**JEU 16**	
	IA<30s>	**A<3>**	**IA<30s>**	**A<4>**
N-gagner	67	6	48	1
N-null	47		60	
T-gagner	55.8%	5%	44%	0.9%
N-pd / N-pl	120 / 120		109 / 110	
TIME	1669.79s/120 =13.9s		1759.64s/110=16s	

Table IV.7

Par cette table, on peut voir que si on définit plus le temps T pour IA, cette méthode va gagner plus.

En comparant le jeu15 (IA<30s> Vs A<3>) avec le jeu 9 (A<4> Vs A<3>) qu'on a fait, le taux de gagner d'IA<30s> (55.8%) est proche de celui de A<4> (58.3%),

le taux de gangner d'A<3> dans le jeu15 est plus bas (5%) que celui dans le jeu 9 (13.3%). Pourtant le temps de jouer une partie dans le jeu15 est plus que celui dans le jeu 9.

De la même façons, on compare le jeu 16 (IA<30s> Vs A<4>) avec le jeu 10 (A<5> Vs A<4>). Le taux de gagner d'IA<30s> (44%) est un peu bas que celui de A<4> (51%), mais le taux de gangner d'A<4> dans le jeu 16 est plus bas (0.9%) que celui dans le jeu 10 (45%). C'est-à-dire quand A<4> joue contre IA<30s>, il n'a pas beaucoup de chance à gagner. En plus le temps de jouer une partie dans le jeu16 (16s) est moins que celui dans le jeu 10 (20.07s).

De la même manière pour ITA, le temps de réflechir T est défini à 10 s, 20s, et 30s.

	JEU 17		**JEU 18**		**JEU 19**	
	ITA<10s>	**A<2>**	**ITA<10s>**	**A<3>**	**ITA<20s>**	**A<3>**
N-gagner	72	13	44	19	47	10
N-null	35		47		53	
T-gagner	60%	10.8%	40%	17.3%	42.7%	9.1%
N-pd / N-pl	120 / 120		110 / 110		110 / 110	
TIME	672.79s/120=5.61s		634.36s/110=5.77s		1061.04s/110=9.65s	

	JEU 20		**JEU 21**	
	ITA<30s>	**A<3>**	**ITA<30s>**	**A<4>**
N-gagner	40	5	47	2
N-null	55		51	
T-gagner	40%	5%	47%	2%
N-pd / N-pl	100 / 100		100 / 100	
TIME	1270.88s/100=12.71s		1467.98s/100=14.68s	

Table IV.8

Par comparer les résultats de la table IV.8 et ceux de la table IV.7, on peut voir ITA n'améliore pas beaucoup l'IA.

5)UCT *U*

UCTEngine a 2 paramètres: *T* et *C*. Pour comparer avec ITA<30s> , une meilleure méthode pour Quarto , le T est défini aussi à 30s. Et le *C* est choisi empiriquement. Parmi les exemples que j'ai fait, 0.65 est une bonne valeur pour C (voir la table IV.9).

	U <C=0.5>	U <C=0.3>	U <C=0.5>	U <C=1>	U <C=0.5>	U <C=0.6>
N-gagner	13	8	11	10	8	14
N-null	29		29		28	
T-gagner	26%	16%	22%	20%	16%	28%
N-pd / N-pl	50 / 50		50 / 50		50 / 50	

	UCT <C=0.6>	UCT <C=0.7>	UCT <C=0.65>	UCT <C=0.6>	UCT <C=0.65>	UCT <C=0.7>
N-gagner	14	7	7	4	9	7
N-null	29		39		34	
T-gagner	28%	14%	14%	8%	18%	14%
N-pd / N-pl	50 / 50		50 / 50		50 / 50	

Table IV.9

Après choisir les paramètres, je lance des jeux entre *UCT* et les méthodes classiques.

	JEU 22		JEU 23		JEU 24	
	U<30s,0.65>	A<3>	U<30s,0.65>	A<4>	U<30s,0.65>	IA<30s>
N-gagner	38	4	29	19	6	33
N-null	58		52		71	
T-gagner	38%	4%	29%	19%	5.5%	30%
N-pd / N-pl	100 / 100		100 / 100		110 /110	
TIME	1678.57s		1822.41s			

Table IV.10

Par la table IV.10, on peut voir que UCT U<30s,0.65> est plus fort que A<3> et A<4>, mais qu'il est moins fort que IA<30s>.

6)AlphaBetaEngineUCT *UA*

C'est une méthode mélangée de l'AlphaBeta et de UCT. Dans la première phase de partie du jeu, la machine utilise la méthode UCT, après elle change à la méthode AlphaBeta. Pour être comparable, les paramètres T =30s comme IA, C=0.65 . Quant à D et N, l'idée est d'essayer de faire AlphaBeta explorer l'arbre jusqu'aux noeuds terminaux.
Je lance des jeux entre UA et IA<30> , qui est la meilleure méthode que j'ai trouvée avant. La table IV.11 montre les résultats de tests.

	JEU 25		JEU 26	
	UA<8,8,30s,0.65>	**IA<30s>**	**UA<9,9,30s,0.65>**	**IA<30s>**
N-gagner	5	44	30	5
N-null	51		65	
T-gagner	5%	44%	30%	5%
N-pd / N-pl	100 / 100		100 / 100	

	JEU 27	
	UA<6,11,30s,0.65>	**IA<30s>**
N-gagner	28	29
N-null	43	
T-gagner	28%	29%
N-pd / N-pl	100 / 100	

Table IV.11

Dans ces 3 jeux (jeu 25,26,27), c'est IA qui commence. Donc , pour UA du jeu 25, quand UA change à utiliser la méthode AlphaBeta, le tablier reste 6 pions et 6 cases vides. Le résultat montre que UA<8,8,30s,0.65> est moins fort que IA<30s>. Pour UA du jeu 26, quand UA change à utiliser la méthode AlphaBeta, le tablier reste 8 pions et 8 cases vides. On peut voir que la méthode UA<9,9,30s,0.65> est fort que IA<30s>. Pour UA du jeu 27, quand UA change à utiliser la méthode AlphaBeta, le tablier reste 10 pions et 10 cases vides. Cest pas possible d'utiliser A<11> en ce moment là, parce que ça va prendre beaucoup de temps. Donc je mettre A<6> ici qui prend le temps plus pertinente. Et le résultat montre que les deux méthodes du jeu 27 ont presque la même capacité.

Conclusion

Pendant ces trois mois de stage, j'ai réalisé le programme du jeu Quarto, appliqué les techniques différentes de la recherhce arborescente, et aussi analysé ces résultats.

Le jeu Quarto est spécial en raison de sa règle particulière. Malgré son petit damier(4x4) et ses 16 pions, il a un arbre de la recherche assez large. Il ne peut pas être un jeu résolu comme Tic-Tac-Toe. C'est-à-dire actuellement la machine ne peut pas jouer ce jeu parfaitement.

Mais la machine peut atteindre un haut niveau par appliquer les techiniques de la recherche arborescente: la techinique classique basé sur l'algorithme *MinMax*, et la technique Monte-Carlo fondée sur la simulation des jeux exemples aléatoires. J'ai étudié 8 méthodes: *StochaEngine (S), MinMaxEngine (M), AlphaBetaEngine (A), IDAlphaBetaEngine (IA), TTAlphaBetaEngine (TA), IDTTAlphaBetaEngine (ITA), UCTEngine (U) et AlphaBetaEngineUCT (UA)*. L'objectif est de trouver une méthode pertinente en condisérant le temps de réfléchir et la capacité.

Après avoir analysé les résultats de mon programme, je peut résumer les 8 methodes comme la suite:

1) StochaEngine est une méthode le plus faible , mais le plus rapide.
2) MinMaxEngine marche bien jusqu'à la profondeur(*D*) 2. Quand *D* est supérieur à2, il n'est plus pratique de l'utilser, car il prend trop de temps pour jouer le premier coup.
3) L'*AlphaBeta* améliore beaucoup le *MinMax*. Mais quand *D*>=5, la machine va prend beaucoup de temps pour jouer au départ de la partie comme le problème de *MinMax*. Par rapport à l'autre profondeurs, *A<4>* est une méthodes pertinente.
4) On applique la *Table de Transposition* pour améliorer l'algorithm *AlphaBeta* . Mais le jeu Quarto n'a la transposition qu'à partir de la profondeur 5, Donc cette amélioration n'est pas très évidente.
5) *Itérative Deepending* contrôle le temps de réfléchir de la machine. 30 second est convenant pour la machine de jouer une partie. Et les résultats montrent que *IA<30>* est fort que *A<4>*.
6) Puisque on sais que la *Table de Transposition* n'améliore pas beaucoup pour Quarto, *ITA<30>* ne peut pas surpasser *IA<30>*.
7) Uct depend des paramètre de *T* et constant *C*. Pour être comparable, on définit *T* =30s. Mais le choix de *C* (*C*=0.65) est empirique. *U<30,0.65>* est fort que *A<4>*, mais il est moins fort que *IA<30>*.

8) Méthode Mélange *UA* est une surprise. *UA<9,9,30s,0.65>* change la méthode au mieu de la partie, et il est fort que *IA<30>*.

Quant au travail futur pour Quarto, je crois qu'il aura encore de l'espace à dévelpper les techniques. Par exemple,comment choisir un meilleur coup au départ ou à la fin de partie? Il existe 'OpenBook' et 'EndBook' qui sont les bases de donnés pour enregistre les coups meilleures dans la première phase et dans la dérnière phase de partie du jeu. Est-ce qu'une évaluation complexe va donner un meilleure résultat pour la techinique classique? Il s'agit de savoir la stratégie de gagner ce jeu Quarto.

Quant au programme, c'est mieux d'améliorer son interface graphique, et de faire l'optimisation sur les codages aussi.

Quel est le niveau de mon programme? C'est difficile à dire parce que il n'y a pas de compétition officielle pour Quarto. J'ai essayé de jouer entre mon programme et les deux programme (Linéo et le programme de Christophe Deprez)dont j'ai parlé dans la partie de l'état de Quarto. Pour cette compétition non officielle, j'ai choisi ma meilleure méthode *UA<9,9,30s,0.65>* , et l'adversaire est choisi le plus haut niveau dans l'option. Le résultat est que mon programme gagne (3-0 vs Linéo, 2-1 vs Christophe Deprez).

En somme, ce stage intéressant m'a aidé de mieux connaître les techiniques de la recherche arborescentes, et la programmation sur le jeu de réflexion.

BIBLIOGRAPHIE SITOGRAPHIE

OEUVRES

- ◣ **Intelligence Artificielle**

  / 2 ème édition Stuart Russell , Peter Norvig

- ◣**Intelligence Artificielle & Informatique Théorique**

  / 2 eme édition CEPADUES,2002, J.-M.ALLIOT

ARTICLES

- *Chaslot,G.,Bouzy,B.,Saito,J-T.,Winands,Mark H.M., Uiterwijk,J.W.H.M., and Jaap van den Herik,H.* **Monte-Carlo Tree Search (2006)**
- *Yizao Wang and Sylvain Gelly.* **Modification of UCT for Monte-Carlo Go with patterns (2006)**
- *Bouzy B..* **Etat de l'art de la programmation des jeux de réflexion (2006)**
- *Coulom,R.* **Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search (2006)** ***Proceedings of the 5th Computers and Games Conference***

SITES

- ◣http://fr.wikipedia.org/wik/TH%C3%A9orie_des_jeux
- ◣http://fr.wikipedia.org/wiki/Quarto
- ◣http://grappa.univ-lille3.fr/~torre/Enseignement/Cours

- ◣http://www.ffothello.org/fede/fede.php
- ◣http://fr.wikipedia.org/wiki/%C3%89lagage_alpha-beta
- ◣http://www.inria.fr/saclay/ressources/zooms/mogo
- ◣http://remi.coulom.free.fr/CrazyStone/
- ◣http://www.alrj.org/docs/algo/ameliorations.php
- ◣http://fr.wikipedia.org/wiki/M%C3%A9thode_de_Monte-Carlo
- ◣http://fr.wikipedia.org/wiki/Lineo

ANNEX

1.Main.cpp (menu)

```
#include <iostream>
#include "damier.h"
#include "time.h"
#include "structur.h"
#include "GenerateMove.h"
#include "MinMaxEngine.h"
#include "AlphaBetaEngine.h"
#include "AlphaBetaEngineUCT.h"
#include "TTAlphaBetaEngine.h"
#include "StochaEngine.h"
#include "IDAlphaBetaEngine.h"
#include "IDTTAlphaBetaEngine.h"
#include "UCTEngine.h"
#include "CheckGameOver.h"
#include "Evaluation.h"
#include "TranspositionTable.h"

using namespace std;

bool CheckCharBelong(char a,char *b)
{for(int i=0;i<4;i++)
      if(a==b[i]) {return true; break;}
  return false;
}
int PreChoisir()
{
int nPion;
cin>>nPion;
return nPion;
}

MOVE HMove()
{
 POSITION to;
 int io,jo;
 MOVE h;
```

```
cin>>io>>jo;
h.to.x=io;
h.to.y=jo;
return h;
}

//humain-humain
void HH()
{
MOVE hA,hB;
Damier damier;
CheckGameOver c;
int premier,deuxieme;
damier.Initial();
damier.Display();
for(int k=0;k<16;k++)
 {

cout<<"\nA choisit un pion pour B:\n";
premier=PreChoisir();
damier.NPionsAJouer=premier;
cout<<"Le pion que A choisit est: pion["<<damier.NPionsAJouer<<"]:"<<damier.pions[premier-
1].caractere<<"\n";

cout<<"\n****************************************\n";
cout<<"B choisit une position   i & j:\n";
hB=HMove();
cout<<"B le déplace à position["<<hB.to.x<<"]"<<"["<<hB.to.y<<"]"<<"\n";
damier.CurrentDamier(hB.to.x,hB.to.y,premier);
damier.Display();
if(c.Check(damier.damier)) {cout<<"Game over,B WIN\n"; break;};
cout<<"\nB choisit un pion pour A:\n";
deuxieme=PreChoisir();
damier.NPionsAJouer=deuxieme;
cout<<"Le pion que B choisit est: pion["<<damier.NPionsAJouer<<"]:"<<damier.pions[deuxieme-1].
caractere<<"\n";
cout<<"\n****************************************\n";

cout<<"A choisit une position   i & j:\n";
hA=HMove();
hA.Score=deuxieme;
```

```
cout<<"A le déplace à position["<<hA.to.x<<"]"<<"["<<hA.to.y<<"]"<<"\n";
damier.CurrentDamier(hA.to.x,hA.to.y,hA.Score);
damier.Display();
if(c.Check(damier.damier)) {cout<<"Game over,A WIN\n"; break;};
}
}

//Humain-Machine
void HM()
{
MOVE h,machine;
Damier damier;
CheckGameOver c;
MinMaxEngine m;
AlphaBetaEngine n;
AlphaBetaEngineUCT nu;
TTAlphaBetaEngine t;
IDTTAlphaBetaEngine dt;
IDAlphaBetaEngine d;
UCTEngine u;
Evaluation e;
GenerateMove currentmove;
clock_t start,finish;
double totaltime,time,timeUCT,constant;
int premier;
int type,depth,E,B;

cout<<"\n*********************************************\n";
cout<<"***CHOISIR LE TYPE DE MACHINE ET PROFONDEUR:***\n";
cout<<"*** 1.MINMAX                              ***\n";
cout<<"*** 2.ALPHABETA                           ***\n";
cout<<"*** 3.ALPHABETA+TT                        ***\n";
cout<<"*** 4.ALPHABETA+TT+ID                     ***\n";
cout<<"*** 5.Iterative Deepening                 ***\n";
cout<<"*** 6.UCT                                 ***\n";
cout<<"*** 7.UCT+ALPHABETA                       ***\n";
cout<<"*********************************************\n";
cout<<"INPUT: Type  &  Depth & Eversion(1,2..) &BestList(1/0) &TimeID &TimeUCT & ParamètreUCT\
n";
cin>>type>>depth>>E>>B>>time>>timeUCT>>constant;
damier.Initial();
```

```
damier.Display();
for(int k=0;k<16;k++)
 {
cout<<"\nHumain choisit un pion pour Machine:\n";
premier=PreChoisir();
damier.NPionsAJouer=premier;
cout<<"Le pion que Humain choisit est: pion["<<damier.NPionsAJouer<<"]:"<<damier.pions[premier
1].caractere<<"\n";
cout<<"\n***************************************\n";
start=clock();
switch(type)
{
case 1:{machine=m.SearchGoodMove(damier,depth,E,B);break;}
case 2:{machine=n.SearchGoodMove(damier,depth,E,B);break;}
case 3:{machine=t.SearchGoodMove(damier,depth,E,B);break;}
case 4:{machine=dt.SearchGoodMove(damier,time,E);break;}
case 5:{machine=d.SearchGoodMove(damier,time,E);break;}
case 6:{machine=u.SearchGoodMove(damier,timeUCT,constant);break;}
case 7:{machine=nu.SearchGoodMove(damier,depth,E,B,timeUCT,constant);break;}
default: machine=n.SearchGoodMove(damier,2,1,1);
}
finish=clock();
totaltime=(double)(finish-start)/CLOCKS_PER_SEC;
time=time-totaltime;//pour ID
cout<<"\nLe temps pour MACHINE est "<<totaltime<<" seconds!"<<endl;

cout<<"Machine le déplace à position["<<machine.to.x<<"]"<<"["<<machine.to.y<<"]"<<"\n";
damier.CurrentDamier(machine.to.x,machine.to.y,machine.PreScore);
damier.Display();
if(c.Check(damier.damier)) {cout<<"Game over, Machine WIN\n"; break;}
damier.NPionsAJouer=machine.Score;
cout<<"Le pion que machine choisit est: pion["<<machine.Score<<"]:"<<damier.pions[machine.Scor
e-1].caractere<<"\n";
cout<<"\n***************************************\n";

cout<<"Humain choisit une position       i & j:\n";
h=HMove();
h.Score=machine.Score;
cout<<"Humain le déplace à position["<<h.to.x<<"]"<<"["<<h.to.y<<"]"<<"\n";
damier.CurrentDamier(h.to.x,h.to.y,h.Score);
damier.Display();
```

```
if(c.Check(damier.damier)) {cout<<"Game over, Humain WIN\n"; break;};
}
}

//Machine-Machine
void MM()
{UCTEngine u;
 Damier damier;
 MinMaxEngine m;
 AlphaBetaEngine n;
AlphaBetaEngineUCT nu;
 TTAlphaBetaEngine t;
 IDAlphaBetaEngine d;
 StochaEngine s;
IDTTAlphaBetaEngine dt;
 CheckGameOver c;
 MOVE machine,machine1;
 Evaluation e;
 int m1,m2,peace,k;//calculer la fois de gagner pour 2 machines.
 int countW,countB,countW1,countB1;
 int premier;
 clock_t start,finish,startALL,finishALL;
 double totaltime,totaltimeALL,timeIDA,timeIDB,timeBKA,timeBKB,timeCKA,timeCKB,timeUCTA,timeUC
TB,constantA,constantB;
 GenerateMove currentmove;
 m1=0;m2=0;peace=0;
 bool finished=0;
 int fois;
 CTranspositionTable ta;
typedef struct _winlose
{
    unsigned int checkID;
   int result;
}WinLose;

//CHOSIR MACHINEA
int typeA,depthA,EA,BA;
cout<<"\n**********************************************\n";
cout<<"***CHOISIR LE TYPE DE MACHINE-A ET PROFONDEUR:*\n";
cout<<"*** 1.MINMAX                              ***\n";
cout<<"*** 2.ALPHABETA                           ***\n";
```

```
cout<<"*** 3.ALPHABETA+TT                              ***\n";
cout<<"*** 4.Iterative Deepening                       ***\n";
cout<<"*** 5.STOCHASITQUE                              ***\n";
cout<<"*** 6.ALPHABETA+TT+ID                           ***\n";
cout<<"*** 7.UCT                                       ***\n";
cout<<"*** 8.UCT+ALPHABETA                             ***\n";
cout<<"***********************************************\n";
cout<<"INPUT: TypeA  &  DepthA & Eversion(1,2..) &BestList(1/0) &TimeID &TimeUCT & Param猫treUCT\n";
cin>>typeA>>depthA>>EA>>BA>>timeIDA>>timeUCTA>>constantA;
timeBKA=timeIDA;
timeCKA=timeUCTA;
//CHOSIR MACHINEB
int typeB,depthB,EB,BB;
cout<<"\n-----------------------------------------------\n";
cout<<"---CHOISIR LE TYPE DE MACHINE-B ET PROFONDEUR:-\n";
cout<<"--- 1.MINMAX                                    ---\n";
cout<<"--- 2.ALPHABETA                                 ---\n";
cout<<"--- 3.ALPHABETA+TT                              ---\n";
cout<<"--- 4.Iterative Deepening                       ---\n";
cout<<"--- 5.STOCHASITQUE                              ---\n";
cout<<"--- 6.ALPHABETA+TT+ID                           ---\n";
cout<<"--- 7.UCT                                       ---\n";
cout<<"--- 8.UCT+ALPHABETA                             ---\n";
cout<<"-----------------------------------------------\n";
cout<<"INPUT: TypeB  &  DepthB & Eversion(1,2..) &BestList(1/0) &TimeID &TimeUCT & Param猫treUCT\n";
cin>>typeB>>depthB>>EB>>BB>>timeIDB>>timeUCTB>>constantB;
timeBKB=timeIDB;
timeCKB=timeUCTB;
//CHOISIR FOIS DE JOUER
cout<<"INPUT: Fois de jouer\n";
cin>>fois;
WinLose checkID[fois];
startALL=clock();
for(int q=0;q<fois;q++)
{
timeIDA=timeBKA;timeIDB=timeBKB;
timeUCTA=timeCKA;timeUCTB=timeCKB;
time_t grain;
srand(time(&grain));
```

```
premier=1+rand()%16;//choisir le premier pion stochastique

checkID[q].checkID=premier;
damier.Initial();
damier.Display();
damier.NPionsAJouer=premier;
cout<<"Le premier pion choisi est: pion["<<damier.NPionsAJouer<<"]:"<<damier.pions[premier-1].
caractere<<"\n";

do
 {
//MACHINE A
cout<<"\n****************************************\n";
k=0;
for(int i=0;i<16;i++) if(damier.pions[i].position.x==0) k++;
if(k==0) {checkID[q].checkID=checkID[q].checkID+ta.CalculateInitHashKey(damier);checkID[q].res
ult=1;peace++;break;}

start=clock();
switch(typeA)
{
case 1:{machine1=m.SearchGoodMove(damier,depthA,EA,BA);break;}
case 2:{machine1=n.SearchGoodMove(damier,depthA,EA,BA);break;}
case 3:{machine1=t.SearchGoodMove(damier,depthA,EA,BA);break;}
case 4:{machine1=d.SearchGoodMove(damier,timeIDA,EA);break;}
case 5:{machine1=s.SearchGoodMove(damier);break;}
case 6:{machine1=dt.SearchGoodMove(damier,timeIDA,EA);break;}
case 7:{machine1=u.SearchGoodMove(damier,timeUCTA/CONSTANTID,constantA);break;}
case 8:{machine1=nu.SearchGoodMove(damier,depthA,EA,BA,timeUCTA/CONSTANTID,constantA);break;}
default: machine1=n.SearchGoodMove(damier,2,1,0);
}
finish=clock();
totaltime=(double)(finish-start)/CLOCKS_PER_SEC;
timeIDA=timeIDA-timeIDA/CONSTANTID;
timeUCTA=timeUCTA-timeUCTA/CONSTANTID;
cout<<"\nLe temps pour MachineA est "<<totaltime<<" seconds!"<<endl;

cout<<"MachineA le d茅place □  position["<<machine1.to.x<<"]"<<"["<<machine1.to.y<<"]"<<"\n";
damier.CurrentDamier(machine1.to.x,machine1.to.y,machine1.PreScore);
damier.Display();
if(c.Check(damier.damier)) {checkID[q].checkID=checkID[q].checkID+ta.CalculateInitHashKey(dami
er);cout<<"Game over machineA win\n";cout<<"***************************************\n";
```

```
 m1++;
checkID[q].result=2;break;}
else{
k=0;
for(int i=0;i<16;i++) if(damier.pions[i].position.x==0) k++;
if(k==0) {checkID[q].checkID=checkID[q].checkID+ta.CalculateInitHashKey(damier);checkID[q].res
ult=1;peace++;break;}
}
damier.NPionsAJouer=machine1.Score;
cout<<"Le pion que MachineA choisit est: pion["<<machine1.Score<<"]:"<<damier.pions[machine1.S
core-1].caractere<<"\n";
cout<<"\n****************************************\n";



//MACHINE B
k=0;
for(int i=0;i<16;i++) if(damier.pions[i].position.x==0) k++;
if(k==0) {checkID[q].checkID=checkID[q].checkID+ta.CalculateInitHashKey(damier);checkID[q].res
ult=1;peace++;break;}
start=clock();
switch(typeB)
{
case 1:{machine=m.SearchGoodMove(damier,depthB,EB,BB);break;}
case 2:{machine=n.SearchGoodMove(damier,depthB,EB,BB);break;}
case 3:{machine=t.SearchGoodMove(damier,depthB,EB,BB);break;}
case 4:{machine=d.SearchGoodMove(damier,timeIDB,EB);break;}
case 5:{machine=s.SearchGoodMove(damier);break;}
case 6:{machine=dt.SearchGoodMove(damier,timeIDB,EB);break;}
case 7:{machine=u.SearchGoodMove(damier,timeUCTB/CONSTANTID,constantB);break;}
case 8:{machine1=nu.SearchGoodMove(damier,depthB,EB,BB,timeUCTB/CONSTANTID,constantB);break;}
default: machine=n.SearchGoodMove(damier,2,1,0);
}
finish=clock();
totaltime=(double)(finish-start)/CLOCKS_PER_SEC;
timeIDB=timeIDB-timeIDB/CONSTANTID;
timeUCTB=timeUCTB-timeUCTB/CONSTANTID;
cout<<"\nLe temps pour MachineB est "<<totaltime<<" seconds!"<<endl;

cout<<"MachineB le d茅place □  position["<<machine.to.x<<"]"<<"["<<machine.to.y<<"]"<<"\n";
damier.CurrentDamier(machine.to.x,machine.to.y,machine.PreScore);
damier.Display();
```

```
if(c.Check(damier.damier)) {checkID[q].checkID=checkID[q].checkID+ta.CalculateInitHashKey(dami
er);cout<<"Game over machineB win\n";cout<<"**************************************\n";m2++;ch
eckID[q].result=3; break;}
else{
k=0;
for(int i=0;i<16;i++) if(damier.pions[i].position.x==0) k++;
if(k==0) {checkID[q].checkID=checkID[q].checkID+ta.CalculateInitHashKey(damier);checkID[q].res
ult=1;peace++;break;}
}
damier.NPionsAJouer=machine.Score;
cout<<"Le pion que MachineB choisit est: pion["<<machine.Score<<"]:"<<damier.pions[machine.Sco
re-1].caractere<<"\n";
cout<<"\n**************************************\n";

}while(!finished);
}
finishALL=clock();
totaltimeALL=(double)(finishALL-startALL)/CLOCKS_PER_SEC;
cout<<"machineA("<<typeA<<" "<<depthA<<" "<<" "<<EA<<" "<<BA<<" "<<timeBKA<<")"<<" win "<<m1<
<" times\n";
cout<<"machineB("<<typeB<<" "<<depthB<<" "<<" "<<EB<<" "<<BB<<" "<<timeBKB<<")"<<" win "<<m2<
<" times\n";
cout<<"peace "<<peace<<" times\n";
cout<<"\nLe temps total est "<<totaltimeALL<<" seconds!"<<endl;

for(int i=0;i<fois;i++)
for(int j=i+1;j<fois;j++)
   if(checkID[i].checkID==checkID[j].checkID) checkID[j].checkID=0;

int j=0;
m1=0;m2=0,peace=0;
for(int i=0;i<fois;i++)
if(checkID[i].checkID!=0)
{
if(checkID[i].result==1) {peace++;j++;}
if(checkID[i].result==2) {m1++;j++;}
if(checkID[i].result==3) {m2++;j++;}
}
cout<<"number of jeux  ="<<j<<"\n";
cout<<"machineA("<<typeA<<" "<<depthA<<" "<<" "<<EA<<" "<<BA<<" "<<timeBKA<<" "<<timeCKA<<" "<
<constantA<<")"<<" win "<<m1<<" times\n";
```

```
cout<<"machineB("<<typeB<<" "<<depthB<<" "<<" "<<EB<<" "<<BB<<" "<<timeBKB<<" "<<timeCKB<<" "<
<constantB<<")"<<" win "<<m2<<" times\n";
cout<<"peace "<<peace<<" times\n";

}

int main()
{
int choix;
char goon;
do{
cout<<"==============================================\n";
cout<<"=== CHOISIR                                ===\n";
cout<<"=== 1.Humain vs Humain                     ===\n";
cout<<"=== 2.Humain vs Machine                    ===\n";
cout<<"=== 3.Machine vs Machine                   ===\n";
cout<<"==============================================\n";
cout<<"INPUT: \n";
cin>>choix;
switch(choix)
{
case 1:{HH();break;}
case 2:{HM();break;}
case 3:{MM();break;}
default: MM();
}
cout<<"==Continued? (y/n)\n";
cin>>goon;
}while(goon=='y');
return 0;
}
```

2.TranspositionTable.cpp

```
#include "damier.h"
#include "structur.h"
#include "TranspositionTable.h"

#ifdef _DEBUG

#undef THIS_FILE

static char THIS_FILE[]=__FILE__;
```

```
#define new DEBUG_NEW
#endif

long long rand64(void)
{       return  rand()  ^  ((long  long)rand()  <<  15)  ^  ((long  long)rand()  <<  30)  ^  ((long
long)rand() << 45) ^ ((long long)rand() << 60);
}
long rand32(void)
{    return rand() ^ ((long)rand() << 15) ^ ((long)rand() << 30);
}

CTranspositionTable::CTranspositionTable()
{
    InitializeHashKey();
}

CTranspositionTable::~CTranspositionTable()
{
    delete m_pTT[0];
    delete m_pTT[1];
}

void CTranspositionTable::InitializeHashKey()
{
    int i,j,k;
    num=0;
    srand( (unsigned)time( NULL ));

for (k = 0; k < NPION; k++)
    for (i = 0; i < 17; i++)

            {    m_nHashKey32[k][i] = rand32();
                 m_ulHashKey64[k][i] = rand64();
            }
    m_pTT[0] = new HashItem[1024*1024*8];
    m_pTT[1] = new HashItem[1024*1024*8];
}
unsigned int CTranspositionTable::CalculateInitHashKey(Damier damier)
{
    int i;
       int x,y;
```

```
    m_HashKey32 = 0;
    m_HashKey64 = 0;
    for (i = 0; i < NPION; i++)
        {  if(damier.pions[i].position.x!=0)
                        {   x=damier.pions[i].position.x-1;y=damier.pions[i].position.y-1;
                m_HashKey32 = m_HashKey32 ^ m_nHashKey32[i][x*4+y];
                m_HashKey64 = m_HashKey64 ^ m_ulHashKey64[i][x*4+y];
            }
        }
           m_HashKey32 = m_HashKey32 ^ m_nHashKey32[damier.NPionsAJouer-1][16];
                m_HashKey64 = m_HashKey64 ^ m_ulHashKey64[damier.NPionsAJouer-1][16];
return m_HashKey32;
}

void CTranspositionTable::Hash_MakeMove(MOVE move)
{
        int x;
    x=(move.to.x-1)*4+move.to.y-1;
    m_HashKey32  =  m_HashKey32  ^  m_nHashKey32[move.PreScore-1][x]  ^  m_nHashKey32[move.Score-
1][16];
    m_HashKey64 = m_HashKey64 ^ m_ulHashKey64[move.PreScore-1][x] ^ m_ulHashKey64[move.Score-
1][16];
 }

void CTranspositionTable::Hash_UnMakeMove(MOVE move)
{
    int x;
    x=(move.to.x-1)*4+move.to.y-1;
    m_HashKey32  =  m_HashKey32  ^  m_nHashKey32[move.PreScore-1][x]  ^  m_nHashKey32[move.Score-
1][16];
    m_HashKey64 = m_HashKey64 ^ m_ulHashKey64[move.PreScore-1][x] ^ m_ulHashKey64[move.Score-
1][16];
}

int CTranspositionTable::LookUpHashTable(int alpha, int beta, int depth,int TableNo)
{
    int x;
    HashItem * pht;

    x = m_HashKey32 %(1024*1024*8);
    pht = &m_pTT[TableNo][x];
```

```
    if (pht->depth >= depth && pht->checksum == m_HashKey64)
    {
        switch (pht->entry_type)
        {
        case exact:
            { if(depth>=0)  return pht->eval;}
        case lower:
            if (pht->eval >= beta)
                {if(depth>=0) return (pht->eval);}
            else
                break;
        case upper:
            if (pht->eval <= alpha)
                {if(depth>=0) return (pht->eval);}
            else
                break;
        }
    }
    return 66666;
}
void CTranspositionTable::LookAtTable()
{
    int x,i=0,j=0;
    HashItem * pht0;
HashItem * pht1;

for(x=0;x<1024*1024*8;x++)
{
 pht0 = &m_pTT[0][x];
if(pht0->checksum!=0)  i++;
}

for(x=0;x<1024*1024*8;x++)
{
 pht1 = &m_pTT[1][x];
if(pht1->checksum!=0)  j++;
}
std::cout<<"  i="<<i<<"  j="<<j<<"  num="<<num<<"\n";
}
void  CTranspositionTable::EnterHashTable(ENTRY_TYPE  entry_type,  int  eval,  int  depth,int
```

```
TableNo)
{
    int x;
    HashItem * pht;
    x = m_HashKey32 %(1024*1024*8);//20bits
        pht = &m_pTT[TableNo][x];
       if(pht->test==100&&pht->checksum == m_HashKey64) num++;
    pht->checksum = m_HashKey64;
    pht->entry_type = entry_type;
    pht->eval = eval;
    pht->depth = depth;
pht->test = 100;
 }
```

UCTTT.cpp

```
#include "damier.h"
#include "structur.h"
#include "UCTTT.h"
#ifdef _UCTDEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif
long long UCTrand64(void)

{    return rand() ^ ((long long)rand() << 15) ^ ((long long)rand() << 30) ^
        ((long long)rand() << 45) ^ ((long long)rand() << 60);
}
long UCTrand32(void)
{    return rand() ^ ((long)rand() << 15) ^ ((long)rand() << 30);
}

UCTTT::UCTTT()
{
    InitializeHashKey();
}
UCTTT::~UCTTT()
{
```

```
    delete m_pTT;
}
void UCTTT::InitializeHashKey()
{
    int i,j,k;
UCTHashItem * pht1;
    num=0;
    srand( (unsigned)time( NULL ));

for (k = 0; k < NPION; k++)
    for (i = 0; i < 17; i++)
            {
                m_nHashKey32[k][i] = UCTrand32();
                m_ulHashKey64[k][i] = UCTrand64();
            }
    m_pTT = new UCTHashItem[1024*1024];
for(int x=0;x<1024*1024;x++)
{
 pht1 = &m_pTT[x];
pht1->eval = 0;
pht1->no = 0;
pht1->win = 0;
pht1->lose = 0;
pht1->neutre = 0;
}
}
unsigned int UCTTT::CalculateInitHashKey(Damier damier)
{
    int i;
        int x,y;
    m_HashKey32 = 0;
    m_HashKey64 = 0;
    for (i = 0; i < NPION; i++)
        {  if(damier.pions[i].position.x!=0)
                        {   x=damier.pions[i].position.x-1;y=damier.pions[i].position.y-1;
                m_HashKey32 = m_HashKey32 ^ m_nHashKey32[i][x*4+y];
                m_HashKey64 = m_HashKey64 ^ m_ulHashKey64[i][x*4+y];
            }
        }
           m_HashKey32 = m_HashKey32 ^ m_nHashKey32[damier.NPionsAJouer-1][16];
                m_HashKey64 = m_HashKey64 ^ m_ulHashKey64[damier.NPionsAJouer-1][16];
```

```
 return m_HashKey32;
}

void UCTTT::Hash_MakeMove(MOVE move)
{
         int x;
    x=(move.to.x-1)*4+move.to.y-1;
    m_HashKey32 = m_HashKey32 ^ m_nHashKey32[move.PreScore-1][x] ^ m_nHashKey32[move.Score-1][16];
    m_HashKey64 = m_HashKey64 ^ m_ulHashKey64[move.PreScore-1][x] ^ m_ulHashKey64[move.Score-1][16];
}

void UCTTT::Hash_UnMakeMove(MOVE move)
{
    int x;
    x=(move.to.x-1)*4+move.to.y-1;
    m_HashKey32 = m_HashKey32 ^ m_nHashKey32[move.PreScore-1][x] ^ m_nHashKey32[move.Score-1][16];
    m_HashKey64 = m_HashKey64 ^ m_ulHashKey64[move.PreScore-1][x] ^ m_ulHashKey64[move.Score-1][16];
}

int UCTTT::LookUpHashTable(unsigned int key32)
{
    int x;
    UCTHashItem * pht;
    x = m_HashKey32 %(1024*1024);
    pht = &m_pTT[x];
    if (pht->checksum == m_HashKey64)  return 1;
    return 66666;
}

int UCTTT::NombreNode(unsigned int key32)
{
    int x,y;
    UCTHashItem * pht;
x = key32 %(1024*1024);
    pht = &m_pTT[x];
y=pht->win+pht->lose+pht->neutre;
    return y;
```

```
}

int UCTTT::NombreWin(unsigned int key32)
{
    int x;
    UCTHashItem * pht;
    x = key32 %(1024*1024);
    pht = &m_pTT[x];
    return pht->win;
}

int UCTTT::NombreLose(unsigned int key32)
{   int x;
    UCTHashItem * pht;
    x = key32 %(1024*1024);
    pht = &m_pTT[x];
    return pht->lose;
}

int UCTTT::NombreNeutre(unsigned int key32)
{
    int x;
    UCTHashItem * pht;
    x = key32 %(1024*1024);
    pht = &m_pTT[x];
    return pht->neutre;
}

double UCTTT::ValueNode(unsigned int key32)
{
    int x;
    UCTHashItem * pht;
    x = key32 %(1024*1024);
    pht = &m_pTT[x];
    return pht->eval;
}

void UCTTT::EnterHashTable(unsigned int key32)
{
    int x;
    UCTHashItem * pht;
```

```
    x = m_HashKey32 %(1024*1024);//20bits
      pht = &m_pTT[x];

    pht->checksum = m_HashKey64;
pht->no = 1;
 }
void UCTTT::UpdateValue(unsigned int key32,int eval,int depth)
{
    int x;
    UCTHashItem * pht;
int turn=(depth)%2;
    x = key32 %(1024*1024);//20bits
    pht = &m_pTT[x];
pht->no++;
if(turn==1)
{ pht->eval = pht->eval+eval;}
else {pht->eval = pht->eval-eval;}
if(eval==1) {/*pht->no++;*/pht->win++;}
if(eval==0) {/*pht->no++;*/pht->neutre++;}
if(eval==-1) {/*pht->no++;*/pht->lose++;}
}
```

4.MinMax.cpp

```
#include "structur.h"
#include "MinMaxEngine.h"
#include <iostream>
#include "stdio.h"

using namespace std;

MinMaxEngine::MinMaxEngine(){};

MinMaxEngine::~MinMaxEngine(){};

MOVE MinMaxEngine::SearchGoodMove(Damier damier,int depth,int ENumber,int BestList)
{int num;
 MOVE p;
 CurrentDamier=damier;
ID=0;
```

```
ENo=ENumber;
SearchDepth=depth;
MaxDepth=SearchDepth;
NodeNumber=0;
//cout<<"count w b="<<countW<<" "<<countB<<endl;
bestmove.to.x=-1;
bestmove.to.y=-1;
cout<<"MinMax:SEARCHDEPTH="<<SearchDepth<<endl;
MinMax(MaxDepth);

cout<<"Node Number="<<NodeNumber<<endl;

srand( (unsigned)time( NULL ));
num=1+rand()%BestMoveNumber;
cout<<"MoveNumber="<<num<<endl;

cout<<"bestmove="<<"  "<<BestMoveList[num].move.to.x<<BestMoveList[num].move.to.y<<endl;

if(BestList==1)
return BestMoveList[num].move;
else
return bestmove;
};

int MinMaxEngine::ChoisirPion(char * damier[NUMBER][NUMBER])
{int i;
 i=1+rand()%16;
 return i;
};

int MinMaxEngine::MinMax(int depth)
{
 int i,j,over,count,val;
 int best=-99990;
  int test;

 over=IsGameOver(CurrentDamier,depth);
 if(over!=0) {if(depth!=0) ID=0;return over;}
 if (depth==0&&CEval.Evaluate(CurrentDamier,ENo)==current) {ID=1;}
 if(depth<=0) {test=CEval.Evaluate(CurrentDamier,(SearchDepth+1-depth)%2);
                     return test; }
```

```
 count = CMg.CreatePossibleMove(CurrentDamier, depth);

 for(i=0;i<count;i++)
 {
  MakeMove(CMg.movelist[depth][i]);
  val=-MinMax(depth-1);
  UnMakeMove(CMg.movelist[depth][i]);

  if(val>best){ best=val; //cout<<"best= "<<best<<"  ";
  if(depth==MaxDepth) {bestmove=CMg.movelist[depth][i];
                                  BestMoveNumber=1;current=best;
                         BestMoveList[BestMoveNumber].move=bestmove;BestMoveList[BestMoveNumbe
r].eval=best;ID=0;}}

if ((ID==1)&&(val== BestMoveList[1].eval)&& (depth==MaxDepth))
{  BestMoveNumber++;
BestMoveList[BestMoveNumber].move=CMg.movelist[depth][i];BestMoveList[BestMoveNumber].eval=val
;ID=0;
}

 NodeNumber++;
 }
//cout<<"BEST="<<best<<endl;
return best;
}
```

5.AlphaBetaEngine.cpp

```
#include "structur.h"
#include "AlphaBetaEngine.h"
#include <iostream>

using namespace std;

AlphaBetaEngine::AlphaBetaEngine(){};

AlphaBetaEngine::~AlphaBetaEngine(){};

bool AlphaBetaEngine::EtatValide(Damier damier)
{int k;
```

```
//check for line
  for(int i=0;i<NUMBER;i++)
  {   k=0;
   for(int j=0;j<NUMBER;j++)
          {   if(damier.damier[i][j][0]!='v') k++; }

         if(k>=2) return false;
     }

//check for column
for(int i=0;i<NUMBER;i++)
  {   k=0;
   for(int j=0;j<NUMBER;j++)
          {   if(damier.damier[j][i][0]!='v') k++; }

         if(k>=2) return false;
     }

//check for diagonal
       k=0;
   for(int j=0;j<NUMBER;j++)
          {   if(damier.damier[j][j][0]!='v') k++; }

         if(k>=2) return false;
     k=0;
    for(int j=0;j<NUMBER;j++)
          {   if(damier.damier[j][3-j][0]!='v') k++; }
                    if(k>=2) return false;


  return true;

};





MOVE AlphaBetaEngine::SearchGoodMove(Damier damier,int depth,int ENumber,int BestList)
{int signe;
 MOVE p;
```

```
 CurrentDamier=damier;

StochaEngine s;
SearchDepth=depth;
MaxDepth=SearchDepth;
NodeNumber=0;


//S'il y a moins de 2 pions  ligne,colomn et diagnale, utiliser Stocha. Sinon,AlphaBeta.
if(EtatValide(CurrentDamier))
{cout<<"Use Stochastique"<<" "<<endl;return s.SearchGoodMove(CurrentDamier);}
else
{
cout<<"ALPHA-BETA:SEARCHDEPTH="<<SearchDepth<<endl;
AlphaBeta(MaxDepth,-99999,99999);
cout<<"Node Number="<<NodeNumber<<endl;

cout<<"BEST-Move="<<bestmove.PreScore<<"    ["<<bestmove.to.x<<"]"<<"["<<bestmove.to.y<<"]"<<"
score="<<bestmove.Score<<"\n";

return bestmove;
}
};

int AlphaBetaEngine::AlphaBeta(int depth,int alpha,int beta)
{
 int i,j,over,count,val;

 over=IsGameOver(CurrentDamier,depth);
 if(over!=0) { NodeNumber++;return over;}

 if(depth<=0) { NodeNumber++;return CEval.Evaluate(CurrentDamier,(SearchDepth+1-depth)%2);}

 count = CMg.CreatePossibleMove(CurrentDamier, depth);

 for(i=0;i<count;i++)
 {
  MakeMove(CMg.movelist[depth][i]);
  val=-AlphaBeta(depth-1,-beta,-alpha);
  UnMakeMove(CMg.movelist[depth][i]);
 if (val> alpha)
```

```
        {
            alpha = val;
            if(depth == MaxDepth)
                {bestmove=CMg.movelist[depth][i];
               current=alpha;cout<<"  Current="<<current<<" ";cout<<"  MaxDepth="<<MaxDepth<
<" ";
cout<<"BESTPrescore="<<bestmove.PreScore<<"   ["<<bestmove.to.x<<"]"<<"["<<bestmove.to.y<<"]"<
<"  score="<<bestmove.Score<<"\n";
                          }
               }
        if (val >= beta)  break;
 }

NodeNumber++;
return alpha;
}
```

6.CheckGameOver.cpp

```
#include "CheckGameOver.h"


#include "structur.h"
CheckGameOver::CheckGameOver()
{
}
CheckGameOver::~CheckGameOver()
{
}
bool CheckGameOver::Belong(char a,char *b)
{for(int i=0;i<4;i++)
      if(a==b[i]) {return true; break;}
  return false;
}

bool CheckGameOver::IsLineOk(char *a[4])
{char c[4];
  // std::cout<<a[0]<<"  na  "<<a[1]<<"  nn  "<<a[2]<<"  nn  "<<a[3];
c[0]=a[0][0];
c[1]=a[0][1];
```

```
c[2]=a[0][2];
c[3]=a[0][3];
// std::cout<<c[0]<<"  na  "<<c[1]<<"  nn  "<<c[2]<<"  nn  "<<c[3];
   for(int j=0;j<4;j++)
       if(Belong(c[j],a[1])&&Belong(c[j],a[2])&&Belong(c[j],a[3])) {return true; break;}

return false;
}
bool CheckGameOver::Check(char * damier[NUMBER][NUMBER])
{char *a[4];
  int k;
//check for line
  for(int i=0;i<NUMBER;i++)
  {   k=0;
   for(int j=0;j<NUMBER;j++)
          {   if(damier[i][j][0]=='v') break;
            k++;a[j]=damier[i][j]; }

         if((k==4)&&IsLineOk(a)) return true;
     }

//check for column
for(int i=0;i<NUMBER;i++)
  {   k=0;
   for(int j=0;j<NUMBER;j++)
          {   if(damier[j][i][0]=='v') break;
            k++;a[j]=damier[j][i]; }

         if((k==4)&&IsLineOk(a)) return true;
     }

//check for diagonal
       k=0;
   for(int j=0;j<NUMBER;j++)
          {   if(damier[j][j][0]=='v') break;
            k++;a[j]=damier[j][j]; }

         if((k==4)&&IsLineOk(a)) return true;
     k=0;
    for(int j=0;j<NUMBER;j++)
          {   if(damier[j][3-j][0]=='v') break;
```

```
            k++;a[j]=damier[j][3-j]; }

         if((k==4)&&IsLineOk(a)) return true;
  return false;

}
```

7.Damier.cpp

```
#include <iostream>
#include "damier.h"
#include "structur.h"
using namespace std;

Damier::Damier()
{}

Damier::~Damier()
{}

void Damier::Initial()
{
int i,j;
for(i=0;i<NUMBER;i++)
for(j=0;j<NUMBER;j++)
 damier[i][j]=VIDE;
//definition des 16 pions
pions[0].caractere="1468";
pions[1].caractere="1467";
pions[2].caractere="1458";
pions[3].caractere="1457";
pions[4].caractere="1368";
pions[5].caractere="1367";
pions[6].caractere="1358";
pions[7].caractere="1357";
pions[8].caractere="2468";
pions[9].caractere="2467";
pions[10].caractere="2458";
pions[11].caractere="2457";
pions[12].caractere="2368";
pions[13].caractere="2367";
```

```
pions[14].caractere="2358";
pions[15].caractere="2357";

pions[0].position.x=0;
pions[1].position.x=0;
pions[2].position.x=0;
pions[3].position.x=0;
pions[4].position.x=0;
pions[5].position.x=0;
pions[6].position.x=0;
pions[7].position.x=0;
pions[8].position.x=0;
pions[9].position.x=0;
pions[10].position.x=0;
pions[11].position.x=0;
pions[12].position.x=0;
pions[13].position.x=0;
pions[14].position.x=0;
pions[15].position.x=0;

NPionsAJouer=-1;//le pion qui est pret   jouer
}

void Damier::CurrentDamier(int io,int jo,int nPion)
{

if(nPion==0) damier[io-1][jo-1]=VIDE; // la condition pour vider une position sur damier. Util
iser dans evaluation.cpp pour retourner //l'etat pr茅c茅dant
else {damier[io-1][jo-1]=pions[nPion-1].caractere;
     pions[nPion-1].position.x=io;
     pions[nPion-1].position.y=jo;} //jouer un coup sur damier, actualiser le damier
}
void Damier::Display()
{
int i,j,k=0;
cout<<"       1     2     3     4    [j]\n";
for(i=0;i<NUMBER;i++)
{
 cout<<"    -------------------------\n";
cout<<"  "<<(i+1)<<" |";
for(j=0;j<NUMBER;j++)
```

```
{
if(damier[i][j]==VIDE) {cout<<"     |";
     }
else cout<<damier[i][j]<<" |";

}
cout<<"\n";
}
cout<<"    -------------------------\n";
cout<<" [i]\n";
for(i=0;i<NPION;i++)
{if(i%4==0) cout<<"\n";
 if(pions[i].position.x==0) cout<<"  pion["<<i+1<<"]="<<pions[i].caractere;

}
cout<<"\n";
}
```

8.Evaluation.cpp

```
#include "Evaluation.h"
#include "math.h"
#include "structur.h"
#include "CheckGameOver.h"
#include "damier.h"

Evaluation::Evaluation()
{
}
Evaluation::~Evaluation()
{
}
int Evaluation::Evaluate(Damier damier,int ENumber)
{int score=0;
  int k,pi,pj;
CheckGameOver c;

if(ENumber==1)//machine
{if(c.Check(damier.damier)) {score=-9999; return score;}
```

```
  else //v茅rifier si le pion que machine choisit peut faire humaine gagner ou pas
{
   score=0;
//check for line
  for(int i=0;i<NUMBER;i++)
  {   k=0;
   for(int j=0;j<NUMBER;j++)
          {   if(damier.damier[i][j][0]!='v') k++;
                          else {pi=i;pj=j;}
                           }
          if(k==3) {damier.CurrentDamier(pi+1,pj+1,damier.NPionsAJouer);if(c.Check(damier.dami
er)) score=9999;
                 damier.CurrentDamier(pi+1,pj+1,0);}

    }

//check for column
  for(int i=0;i<NUMBER;i++)
  {   k=0;
   for(int j=0;j<NUMBER;j++)
          {   if(damier.damier[j][i][0]!='v') k++;
                          else {pi=j;pj=i;}
                           }
            if(k==3) {damier.CurrentDamier(pi+1,pj+1,damier.NPionsAJouer);if(c.Check(damier.da
mier)) score=9999;
                          damier.CurrentDamier(pi+1,pj+1,0);}

    }


//check for diagonal
       k=0;
   for(int j=0;j<NUMBER;j++)
{   if(damier.damier[j][j][0]!='v') k++;
                          else {pi=j;pj=j;}
                           }
            if(k==3) {damier.CurrentDamier(pi+1,pj+1,damier.NPionsAJouer);if(c.Check(damier.da
mier)) score=9999;
                        damier.CurrentDamier(pi+1,pj+1,0);}

  k=0;
```

```
    for(int j=0;j<NUMBER;j++)
          {    if(damier.damier[j][3-j][0]!='v') k++;
                          else {pi=j;pj=3-j;}
                           }
            if(k==3) {damier.CurrentDamier(pi+1,pj+1,damier.NPionsAJouer);if(c.Check(damier.da
mier)) score=9999;
                  damier.CurrentDamier(pi+1,pj+1,0);}

return score;
}
}
else //humain
{
if(c.Check(damier.damier)) {score=9999; return score;}
  else //v茅rifier si le pion que humaine choisit peut faire machine gagner ou pas
{
   score=0;
//check for line
  for(int i=0;i<NUMBER;i++)
  {    k=0;
   for(int j=0;j<NUMBER;j++)
          {    if(damier.damier[i][j][0]!='v') k++;
                          else {pi=i;pj=j;}
                           }
          if(k==3) {damier.CurrentDamier(pi+1,pj+1,damier.NPionsAJouer);if(c.Check(damier.dami
er)) score=-9999;
                          damier.CurrentDamier(pi+1,pj+1,0);}

    }
//check for column
  for(int i=0;i<NUMBER;i++)
  {    k=0;
   for(int j=0;j<NUMBER;j++)
          {    if(damier.damier[j][i][0]!='v') k++;
                          else {pi=j;pj=i;}
                           }
            if(k==3) {damier.CurrentDamier(pi+1,pj+1,damier.NPionsAJouer);if(c.Check(damier.da
mier)) score=-9999;
                     damier.CurrentDamier(pi+1,pj+1,0);}

    }
```

```
//check for diagonal
       k=0;
   for(int j=0;j<NUMBER;j++)
{   if(damier.damier[j][j][0]!='v') k++;
                          else {pi=j;pj=j;}
                           }
            if(k==3) {damier.CurrentDamier(pi+1,pj+1,damier.NPionsAJouer);if(c.Check(damier.da
mier)) score=-9999;
                       damier.CurrentDamier(pi+1,pj+1,0);}

     k=0;
    for(int j=0;j<NUMBER;j++)
          {   if(damier.damier[j][3-j][0]!='v') k++;
                          else {pi=j;pj=3-j;}
                           }
            if(k==3) {damier.CurrentDamier(pi+1,pj+1,damier.NPionsAJouer);if(c.Check(damier.da
mier)) score=-9999;
                          damier.CurrentDamier(pi+1,pj+1,0);}

 return score;
}
}
}
```