

Rapport de projet: Gomoku

Quentin Garrido, Antoine G  lin, K  vin Lor

22 f  vrier 2019

Table des mati  res

1	Introduction	2
2	Utilisation	2
3	Structures de donn��es	3
3.1	Repr��sentation du plateau	3
3.2	Pattern	4
3.3	Coup	5
3.4	Arbre des coups	5
4	Choix des coups	6
4.1	Negamax	6
4.2	��lagage alpha-beta	7
5	��valuation d'une position	9
6	Conlusion	11
6.1	R��sultats	11
6.2	Pour aller plus loin	12

1 Introduction

Le but de ce projet est de r  aliser une "intelligence artificielle" pouvant jouer contre un humain au gomoku.

Nous avons choisi d'utiliser un plateau de 8×8 et une victoire avec l'alignement de 5 pions de m  me couleur.

Nous avons programm   en C++ afin d'avoir un langage familier    tout le groupe et qui nous permet d'utiliser certaines repr  sentations pour nos structures de donn  es que nous verrons par la suite.

L'un des membres du groupe s'int  ressant au moteurs d'  checs et ayant une connaissance des algorithmes utilis  s nous avons transpos   ces m  thodes vers le gomoku, qui poss  de les m  me caract  ristique tout en   tant plus simple.

Le projet   t   r  alis      travail   gal par tous les membres du groupe en suivant la r  partition suivante :

- Quentin Garrido : Structures de donn  es et parcours des coups
- Antoine G  lin :   valuation des coups et r  alisation des tests
- K  vin Lor :   valuation des coups et gestion de la m  moire

2 Utilisation

Tout le code source est disponible    l'adresse suivante : <https://github.com/garridoq/gomoku>.

Tout les ex  cutable devraient vous   tre fournis dans le mail et devraient fonctionner sans devoir les recompiler. Dans le cas contraire voici la d  marche    suivre :

Afin de pouvoir compiler les ex  cutable il faut   tre sous Linux et avoir g++ d'install   sur la machine. La norme utilis  e est le C++14 pour tout le programme.

Une fois le code source obtenu il faudra ex  cuter la commande suivante pour compiler les ex  cutable :

```
> make
```

Tout en   tant dans le dossier du code source.

Vous obtiendrez plusieurs ex  cutable de test, ayant un nom comme *test_**. Vous aurez aussi un fichier principal *main* qui vous permettra de jouer contre l'IA.

Pour jouer appelez directement main de la mani  re suivante :

```
> ./main DIFFICULTE
```

Si DIFFICULTE vaut "medium" vous serez en difficult   r  duite (profondeur de recherche 2).

Si DIFFICULTE est absent ou vaut autre chose, vous serez en difficult   maximale (profondeur de recherche 4).

3.2 Pattern

Les patterns seront un   l  ment clef de notre programme car nous permettrons d'  valuer une position. Ils seront eux aussi repr  sent  s par des *bitboards* et nous aurons aussi l'information sur leur *hauteur* et *largeur*. Afin de faciliter le parcours du motif sur le plateau de jeu, nous ferons commencer les motifs le plus en haut    gauche possible du plateau.

			X				
		X					
	X						
X							

0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

FIGURE 2 – Repr  sentation d'un motif de taille 4x4 avec son bitboard associ  

Comme nous pouvons le voir sur cet exemple nous avons un motif de largeur et hauteur 4, plac   le plus en haut    gauche possible qui pourra   tre repr  sent   par un bitboard assez facilement.

Afin de savoir combien de fois un motif est pr  sent dans un bitboard nous utiliserons l'algorithme de parcours suivant :

Algorithm 1 Algorithme de pattern matching

```

1: procedure PATTERN_MATCHING(bitboard, pattern)
2:   count  $\leftarrow$  0
3:   for  $i \leftarrow 0$  to 8 - pattern.height do
4:     for  $j \leftarrow 0$  to 8 - pattern.width do
5:       if pattern & bitboard = pattern then                                 $\triangleright$  & repr  sente un ET logique bit par bit
6:         count  $\leftarrow$  count + 1
7:       end if
8:       pattern  $\leftarrow$  pattern >> 1                                        $\triangleright$  >> est un d  calage de  $n$  bits    droite
9:     end for
10:    pattern  $\leftarrow$  pattern >> pattern.width-1
11:  end for
12:  return count
13: end procedure

```

Cet algorithme va faire glisser le motif sur le plateau et    chaque fois v  rifier si il est pr  sent ou non, en effet si un motif est pr  sent sur le bitboard en faisant un ET logique bit par bit entre les deux nous devrions retrouver notre motif.

3.3 Coup

Le coup va   tre un   l  ment central de notre mod  lisation car il sera utilis   par toutes les parties du programme.

Un coup sera mod  lis   avec les attributs suivants :

- plateau : le plateau de jeu avant que le coup soit jou  
- index : l'indice du bit o   nous jouons un coup
- side : joueur qui r  alise le coup, BLANC ou NOIR
- evaluation : l'  valuation de la position apr  s avoir jou   le coup

Nous obtiendrons le plateau apr  s le coup via une proc  dure qui nous le retournera.

Nous avons choisi de ne pas stocker le plateau apr  s le coup en m  moire directement car cette repr  sentation nous paraissait plus intuitive, et qu'apr  s r  flexion nous n'avons pas d  cel   de diff  rence r  elles entre les deux m  thodes, que ce soit en terme de performance ou de m  moire dans notre impl  mentation finale.

Nous pourrions gagner en m  moire avec notre m  thode en ne stockant le plateau qu'une seule fois et en donnant cette r  f  rence    tous ses enfants dans l'arbre des coups, ce qui nous ferait   conomiser de la m  moire mais qui en pratique n'aurait pas fait de r  elle diff  rence car seule une faible partie de l'arbre est en m  moire    un instant donn  .

3.4 Arbre des coups

L'arbre des coups nous permettra d'obtenir tous les coups jouables depuis une position jusqu'   une profondeur n .

Nous repr  senterons cet arbre par des *noeuds* qui auront les attributs suivants :

- parent : noeud parent, si nous avons besoin de remonter l'arbre des coups
- coup : coup associ   au noeud
- enfants : tous les noeuds ayant des coups r  alisables depuis le coup du noeud courant

En pratique nous ne g  n  rerons pas tout l'arbre jusqu'   la profondeur d'  valuation pour des raisons de co  t en m  moire.

En effet un arbre avec 64 coups possibles au d  part, une profondeur de recherche de 10 et une taille en m  moire de 256bits (taille sous estim  e par rapport    la r  alit  ) demanderait $\frac{64!}{(64-10)!} \cdot \frac{256}{8} = 1,7 \times 10^{19}$ octets de m  moire, soit plus de 10 exaoctets, ce qui est impossible    stocker, que ce soit en m  moire vive ou non.

Nous allons donc g  n  rer les coups lorsque nous en allons en avoir besoin et tirer avantage de la faible dur  e de vie d'une variable locale, ainsi que de l'ordre d'  valuation des coups afin d'avoir seulement un faible nombre de coups charg   en m  moire    un instant donn  .

4 Choix des coups

4.1 Negamax

Afin de choisir parmi les diff  rents coups, la m  thode la plus populaire (et la plus intuitive) est les Minimax.

L'id  e est que nous allons toujours chercher    maximiser notre score en choisissant un coup et que notre adversaire va chercher    minimiser notre score (maximiser le sien). Cela est exploitable car le gomoku est un jeu    somme nulle et ainsi chaque joueur    tout int  r  t    gagner.

En g  n  rant tous les coups jusqu'   une profondeur choisie nous allons successivement vouloir soit maximiser soit minimiser notre score. En pratique nous allons appeler r  cursivement la proc  dure sur tous les coups enfants jusqu'   ce que nous atteignons la fin de l'arbre, et l   seulement nous utiliserons notre fonction d'  valuation de position et le score trouv   sera remont   jusqu'   notre coup actuel afin de d  cider quel coup jouer.

Dans notre cas, nous utiliserons une variant du Minimax, le Negamax qui simplifiera le code, au lieu d'avoir une fonction *Min* et une fonction *Max* nous aurons une seule fonction *Negamax*.

Cette variante s'appuie sur la relation $\max(a, b) = -\min(-a, -b)$ et n  cessite que la fonction d'  valuation renvoie un r  sultat relatif au joueur, tel que $\text{score}(\text{Joueur1}) = -\text{score}(\text{Joueur2})$.

L'algorithme est simplifi   et sera au final :

Algorithm 2 Algorithme du Negamax

```

1: procedure NEGAMAX(node, depth)
2:   if depth = 0 then
3:     return EVALUATE(node.move)
4:   end if
5:   max  $\leftarrow -\infty$ 
6:   for all child in node.children do
7:     score  $\leftarrow$  -NEGAMAX(child, depth - 1)
8:     if score > max then
9:       max  $\leftarrow$  score
10:    end if
11:  end for
12:  return max
13: end procedure

```

Cependant la fonction ne renvoie que le score et pas le coup    choisir, nous allons donc utiliser une autre proc  dure pour l'appel initial ROOTNEGAMAX qui va   tre quasi identique sauf que juste apr  s la ligne 9 nous allons aussi sauvegarder le coup. Cette m  thode nous permet de sauvegarder un coup uniquement si il est jouable directement et de ne pas sauvegarder par m  garde un coup plus bas dans l'arborescence.

Cet algorithme fonctionne tr  s bien mais nous devons quand m  me   valuer un nombre de positions grandissant   norm  ment en fonction de la profondeur :

$$\text{nombre de noeuds} = \frac{(\text{nombre de coups possibles initialement})!}{(\text{nombre de coups possibles initialement} - \text{profondeur})!}$$

Heureusement nous avons une m  thode tr  s efficace pour r  duire le nombre de coups   valu  s : l'  lagage Alpha Beta.

4.2   lagage alpha-beta

L'  lagage alpha beta est un algorithme tr  s puissant qui dans le meilleur des cas va   valuer \sqrt{n} noeuds contre n pour le n  gamax et n dans le pire des cas.

Le principe est le suivant : Un noeud qui ne peut pas contribuer positivement    l'  valuation n'a pas      tre   valu  .

En effet lorsque nous sommes surs que continuer      valuer ne changera pas le r  sultat, autant s'arr  ter. Cela va   tre r  alis   gr  ce    deux variables α et β qui vont former l'intervalle des valeurs qui pourraient contribuer    l'  valuation.

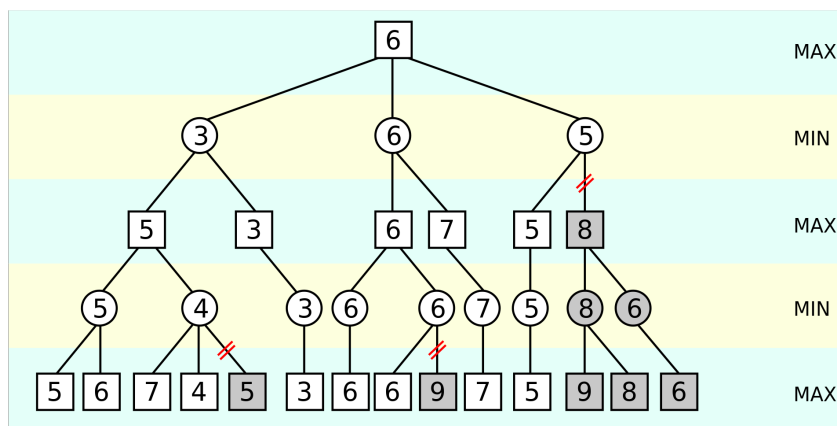


FIGURE 3 – Exemple d'ex  cution de l'algorithme alpha beta. Source : https://commons.wikimedia.org/wiki/File:AB_pruning.svg

Dans cet exemple nous avons trois   lagages qui ont eu lieu (tous les noeuds sont affich  s   valu  s pour la clart  ), de gauche    droite :

Nous cherchons    maximiser la valeur du noeud 5    la profondeur 3 mais    minimiser la valeur de ses enfants. Ainsi pour que le noeud 4 soit utile il faudrait que sa valeur puisse encore d  passer 5 (notre valeur d' α) en explorant ses enfants. Or comme nous minimisons sa valeur nous ne pourrions jamais d  passer 4, et par cons  quent, ainsi il est inutile de continuer    d'  valuer le noeud qui aurait valu 5. Nous avons   conomis   une   valuation ici.

Nous cherchons ensuite    maximiser la valeur du noeud 6    la profondeur 3 en minimisant la valeur de ses enfants. Ainsi le second noeud 6 n'est int  ressant que si il peut d  passer le 6 (notre valeur d' α) d  j   calcul  , or c'est impossible pour la m  me raison que pr  c  demment, donc nous gagnons encore une   valuation.

Enfin, nous cherchons    minimiser le noeud 5    la profondeur 2 tout en maximisant ses enfants. Nous avons d  termin   que sa valeur ne pouvait pas   tre sup  rieure    5, or pour   tre utile il faudrait qu'elle puisse d  passer 6 (notre α) pour remplacer la valeur du noeud 6 de profondeur 1. Ainsi il est inutile de regarder le reste de son sous arbre car cette partie est inutile. Nous gagnerons ici 6   valuations.

Dans ce cas nous aurions   valu   33 noeuds avec le n  gamax et seulement 25 avec l'alpha beta, ce qui repr  sente un gain non n  gligeable. Nous sommes loin de la valeur optimale th  orique mais l'am  lioration est cons  quente et le sera de plus en plus lorsque la taille de l'arbre augmentera.

Dans la m  me id  e que pr  c  demment pour passer du MINIMAX au N  GAMAX nous avons pu adapter la proc  dure pour qu'elle soit correcte dans notre framework n  gamax. L'algorithme est plus court mais pas forc  ment plus simple que la version reposant sur le minimax. Il est le suivant :

Algorithm 3 Algorithme de l'  lagage alpha-beta

```

1: procedure ALPHABETA(node, alpha, beta, depth)
2:   if depth = 0 then
3:     return EVALUATE(node.move)
4:   end if
5:   for all child in node.children do
6:     score  $\leftarrow$  -ALPHABETA(child, -beta, -alpha, depth - 1)
7:     if score  $\geq$  beta then
8:       return beta
9:     end if
10:    if score > alpha then
11:      alpha  $\leftarrow$  score
12:    end if
13:  end for
14:  return alpha
15: end procedure

```

Comme pour le NEGAMAX nous appelons initialement cette proc  dure depuis une version modifi  e qui r  cup  rera le coup apr  s la ligne 11.

Les param  tres α et β seront respectivement initialis  s    $-\infty$ et $+\infty$.

Une fa  on de r  duire le nombre de noeuds   valu  s serait de consid  rer    la gauche de l'arbre (les coups   valu  s en premiers) les coups potentiellement bons en choisissant une heuristique telle que la distance au centre du pion pos   par le coup, car un coup a plus de chance d'  tre bon si il est au centre.

5   valuation d'une position

Tout d'abord, afin de respecter les conditions d'utilisation du framework Negamax, nous devons avoir une fonction d'  valuation asym  trique, elle sera d  finie dans notre cas par :   valuation =   valuation(joueur courant) -   valuation(adversaire), ce qui respecte cette condition.

Notre   valuation sera bas  e sur du pattern matching, avec un tr  s grand nombre de patterns, mais tout d'abord nous devons v  rifier si la partie est termin  e pour donner des scores signifiant la victoire (ou la d  faite).

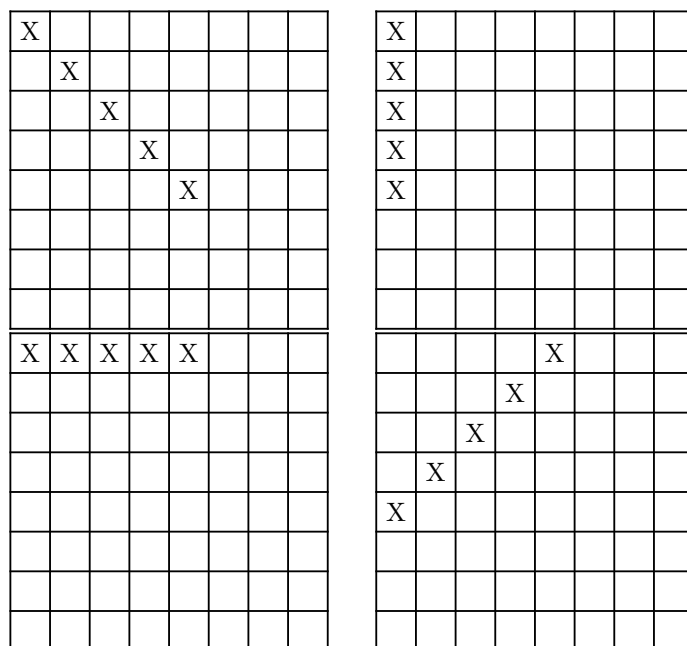


FIGURE 4 – Patterns v  rifi  s pour la victoire ou non

Afin de tester la victoire ou non, nous devons avoir 5 pions align  s, ce qui correspond    savoir si l'un des motifs ci dessus est pr  sent dans le plateau du joueur courant. Dans le cas o   l'un serait pr  sent nous retournerons un score de 100000 qui correspondra    une valeur innatignable, mais pas tout    fait l'infini, nous devons rester plus petit que les valeurs d'infini utilis  es pour α et β lors de l'  lagage alpha beta.

Sinon, nous utiliserons des patternes similaires, notamment des lignes bris  es, des lignes comme pour la victoire mais de longueur inf  rieure. Mais nous regarderons aussi lorsque l'ennemi bloque notre avanc  e, comme par exemple pour une ligne drotie de 3 de longueur bloqu  e des deux c  t  s par notre adversaire. Cependant il faut r  ussir    d  terminer quels motifs sont importants    regarder car plus nous   valuons de motifs sur chaque position, moins nous pourrons   valuer de positions en un temps imparti. Il faut donc faut trouver le juste milieu entre quantit   et qualit   des motifs recherch  s.

De plus certaines zones du plateau sont plus int  ressantes d'un point de vue strat  gique, notamment le centre, ainsi nous pr  f  rons jouer un pion au centre du plateau plut  t que dans un coin. Nous mat  rialisons cette priorit   de placement par la grille coefficient  e suivante :

1	2	3	4	4	3	2	1
2	3	4	5	5	4	3	2
3	4	5	6	6	5	4	3
4	5	6	7	7	6	5	4
4	5	6	7	7	6	5	4
3	4	5	6	6	5	4	3
2	3	4	5	5	4	3	2
1	2	3	4	4	3	2	1

FIGURE 5 – Coefficients relatifs    la position sur le plateau

La valeur d’une case est alors inversement proportionnelle    sa distance euclidienne au centre. Cela va nous permettre de favoriser certaines positions tout en   liminant certaines qui seraient illogiques, par exemple lorsque le plateau est vide, nous ne voulons pas jouer la premi  re case visit  e mais une bonne case, et bien que ce choix soit purement arbitraire il produit des r  sultats coh  rents avec la strat  gie qu’un humain utiliserait.

En combinant les patternes et cette grille nous arrivons    de tr  s bons r  sultats, que ce soit en d  but de partie ou en fin de partie car nous arrivons    la fois      valuer correctement une position sur un plateau presque vide et sur un plateau presque rempli.

Ainsi bien que cette   valuation soit fondamentale pour avoir un bon algorithme, c’est la partie la plus subjective, car nous ne connaissons pas forc  ment la strat  gie optimale. Bien que le jeu ai   t   tr  s   tudi   sur une grille de 15×15 nous sommes sur une grille plus petite et certaines observations ou strat  gies ne sont pas transposables parfaitement.

De plus si nous voullions la meilleur fonction d’  valuation possible (bien qu’aucune ne soit la meilleure de mani  re absolue) il nous faudrait   tudier le jeu plus en profondeur pour voir   merger des patternes et des strat  gies int  ressante, ou,    la mani  re des concepterus d’IA d’  checs, collaborer avec d’excellents joueurs afin de transf  rer leur savoir    la machine.

Nous atteignons bien ainsi une limite de la fonction d’  valuation, elle ne pourra jamais   tre fondamentalement meilleure que nous car nous l’avons programm  , et ainsi elle va imiter notre style de jeu. Ainsi si nous avons programm   une fonction d’  valuation mauvaise, la machine aura beau calculer bien plus rapidement que nous, nous serons toujours capable de gagner. Cependant si elle est bien programm  e (ce qui va   tre dur      valuer), l’avantage de la puissance de calcul de la machine la fera nous dominer.

6 Conclusion

6.1 Résultats

Pour nous mesurer à la machine, nous avons utilisé une profondeur de recherche de 4 et l'élagage alpha beta.

Le premier constat que nous avons pu faire est qu'il est presque impossible de gagner contre la machine, elle arrive bien à prévoir nos coups et notre victoire et nous empêche donc de l'atteindre. Ainsi la grande majorité des parties se sont terminées en égalité ou en victoire de la machine lorsque nous avons commis des erreurs.

Un exemple de partie que nous avons réalisé que nous avons trouvé intéressant est le suivant :

The image displays two 10x10 grids used for a Connect Four game. Each grid has 10 columns and 10 rows, with the bottom row representing the base of the board.

Left Grid (White Wins):

- Row 1: Empty
- Row 2: Empty
- Row 3: Empty
- Row 4: Empty
- Row 5: Empty
- Row 6: Empty
- Row 7: Empty
- Row 8: Empty
- Row 9: 'O' in column 10
- Row 10: 'O' in column 9, 'O' in column 10

Right Grid (Black Wins):

- Row 1: Empty
- Row 2: Empty
- Row 3: 'X' in column 8
- Row 4: Empty
- Row 5: Empty
- Row 6: Empty
- Row 7: Empty
- Row 8: 'X' in column 8
- Row 9: 'X' in column 8, 'O' in column 9, 'O' in column 10
- Row 10: 'X' in column 8, 'O' in column 9, 'O' in column 10

FIGURE 6 – Exemple de début de partie

Dans cette partie nous avons joué les O et la machine les X, comme nous pouvons le voir nous avons vite été stoppés mais pas de la manière la plus évidente. Au lieu de venir se positionner directement devant nous, l'ordinateur a joué une case plus loin, ce qui aura le même résultat pour nous limiter mais qui à moyen terme avantagera l'ordinateur car ses deux pions pourront ainsi être reliés par une ligne droite.

Cette exemple illustre bien l'utilité de calculer en profondeur les coups car cela nous donne des stratégies comme celle ci qui se révèlent très efficaces.

Cependant nous avons pu dans certains cas piéger la machine qui de temps en temps jouait de manière étonnante, en jouant toujours la case correspondant au bit de poids faible du bitboard. Ceci est du à un bug dans notre code mais nous n'avons pas pu le trouver pour le résoudre à temps.

Mais lorsque cela se produit, l'ordinateur pense toujours à nous bloquer pour ne pas que nous gagnions la partie, donc même si ses chances de gagner deviennent quasi-nulles, sa défaite n'est pas garantie.

6.2 Pour aller plus loin

Comme nous avons pu le voir nous avons obtenu un r  sultat tr  s satisfaisant, mais qui pourrait toujours   tre am  lior  .

Tout d'abord la fonction d'  valuation pourrait l'  tre, que ce soit en r  alisant plus de tests pour ajuster les diff  rents motifs/poids, ou en   tudiant encore plus en d  tail les strat  gies possibles pour gagner.

Mais la plupart des am  liorations que nous pourrions r  aliser sont plus li  es    l'impl  mentation des algorithmes qu'aux algorithmes en eux m  mes.

Comme explicit   plus haut, nous pourrions   valuer les positions dans un certain ordre pour r  duire le nombre de noeuds   valu  s par l'  lagage alpha beta.

Nous pourrions aussi parall  liser l'  valuation des coups et le pattern matching afin de b  n  ficier des multiples coeurs pr  sents sur nos machines. Cela nous donnerait un gain de performance consid  rable et nous permettrait d'  valuer plus de positions, afin d'  tre encore meilleur qu'actuellement.

Nous pourrions aussi faire   valuer la profondeur de calcul en fonction de la position, car sur un plateau plus rempli nous pouvons calculer plus de positions en un m  me temps, ce qui permettrait d'  tre encore plus efficace en fin de partie. De mani  re   quivalent, nous n'avons pas forc  ment de regarder tr  s profond  ment en d  but de partie, ce qui nous permettrait alors de gagner du temps.

De mani  re   core plus li  e    l'impl  mentation, nous pourrions optimiser notre usage de la m  moire, limiter le nombre d'op  rations, et profiter encore plus de certaines op  rations qui sont impl  ment  es tr  s efficacement, par exemples les op  rateurs `>>` et `<<` sont impl  ment  s au niveau hardware, ce qui les rends tr  s rapides.

Ainsi la majeure partie des am  liorations des performances seraient du c  t   de la fonction d'  valuation pour augmenter la qualit   de notre algorithme et du c  t   de l'impl  mentation des nos algorithmes pour gagner en vitesse.