

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	6
5	��valuation d'une position	7
6	Conlusion	8
6.1	R��sultats	8
6.2	Pour aller plus loin	8

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 8x8 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 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.

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

Algorithm 2 Algorithme du Negamax

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

4.2   lagage alpha-beta

5   valuation d'une position

X							
	X						
		X					
			X				
				X			
X	X	X	X	X			

X							
X							
X							
X							
X							
				X			
			X				
		X					
	X						
X							

FIGURE 3 – Patterns v  rifi  s pour la victoire ou non

6 Conlusion

6.1 R  sultats

6.2 Pour aller plus loin