Rapport de projet: Gomoku

Quentin Garrido, Antoine Gélin, Kévin Lor

22 février 2019

Table des matières

1 Introduction

Le but de ce projet est de réaliser une "intelligence artificielle" pouvant jouer contre un humain au gomoku.
Nous avons choisi d'utiliser un plateau de $8imes8$ et une victoire avec l'alignement de 5 pions de même couleur.
Nous avons programmé en C++ afin d'avoir un langage familier à tout le groupe et qui nous permet d'utiliser certaines représentations pour nos structures de données que nous verrons par la suite.
L'un des membres du groupe s'intéressant au moteurs d'échecs et ayant une connaissance des algorithmes utilisés nous avons transposé ces méthodes vers le gomoku, qui possède les même caractéristique en étant plus simple.

Le projet à été réalisé à travail égal par tous les membres du groupe en suivant la répartition suivante :
— Quentin Garrido : Structures de données et parcours des coups
— Antoine Gélin : Évaluation des coups et réalisation des tests
— Kévin Lor : Évaluation des coups et gestion de la mémoire

2 Utilisation

Tout le code source est disponible à l'adresse suivante : *https ://github.com/garridoq/gomoku.*

Tout les éxécutables devraient vous être fournis dans le mail et devraient fonctionner sans devoir les recompiler. Dans le cas contraire voici la démarche à suivre :

Afin de pouvoir compiler les exécutables il faut être sous Linux et avoir g++ d'installé sur la machine. La norme utilisée est le C++14 pour tout le programme.

Une fois le code source obtenu il faudra exécuter la commande suivante pour compiler les exécutables :

> make

Tout en étant dans le dossier du code source.

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

3 Structures de données

3.1 Représentation du plateau

Pour représenter le plateau nous n'avons pas utilisé de tableau mais des *bitboards* qui sont la représentation d'un plateau avec chaque case repréenté par un bit, ainsi si elle est occupée le bit vaudra 1 et 0 sinon. Le bit de poids fort sera le coin en haut à gauche de notre plateau et celui de poids faible le coin en bas à droite. Nous avons alors la représentation suivante pour un plateau de 8×8 :

					X		
		X	X				
				X			
					X		

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

				O			
	O				O		
		O	O				

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

Figure 1 – Représentation d'un plateau avec les bitboards associés

Ainsi nous gagnons en mémoire, une position ne nécessite plus que 128bits pour être stockée et nous béneficions de l'implémentation en hardware du décalage des nombres, ce qui nous fera gagner de la rapidité à l'évaluation des coups.

C'est ce choix de représentation qui a motivé notre choix d'utiliser le C++ pour le projet. En effet il est un des rares langages qui nous permet d'accéder à chaque bits individuellement d'un nombre et de réaliser des opérations logiques bit par bit.
Pour représenter un plateau de nxn il nous faut n^2 bits d'où le choix du plateau 8×8 car un entier au delà de 64bits est plus dur à représenter en C++.

Cette réprésentation va aussi simplifier l'évaluation des coups en elle même car il sera plus simple de réaliser un parcours de motif (pattern) en décalant les bits de ce dernier.

3.2 Pattern

Les patterns seront un élément clef de notre programme car nous permettrons d'évaluer une position. Ils seront eux aussi représentés 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 à 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 – Représentation d'un motif de taille 4x4 avec son bitboard associé

Comme nous pouvons le voir sur cet exemple nous avons un motif de largeur et hauteur 4, placé le plus en haut à gauche possible qui pourra être représenté par un bitboard assez facilement.

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

Algorithm 1 Algorithme de pattern matching

```
procedure PATTERN_MATCHING(bitboard, pattern)
    count ← 0
    for i ← 0 to 8 − pattern.height do
        for j ← 0 to 8 − pattern.width do
            if pattern & bitboard = pattern then        ▷ & représente un ET logique bit par bit
                count ← count + 1
            end if
            pattern ← pattern >> 1                      ▷ >> est un décalage de n bits à droite
        end for
        pattern ← pattern >> pattern.width-1
    end for
    return count
end procedure
```

Cet algorithme va faire glisser le motif sur le plateau et à chaque fois vérifier si il est présent ou non, en effet si un motif est présent 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 être un élément central de notre modélisation car il sera utilisé par toutes les parties du programme.
Un coup sera modélisé avec les attributs suivants :
- plateau : le plateau de jeu avant que le coup soit joué
- index : l'indice du bit où nous jouons un coup
- side : joueur qui réalise le coup, BLANC ou NOIR
- evaluation : l'évaluation de la position après avoir joué le coup

Nous obtiendrons le plateau après le coup via une procédure qui nous le retournera.
Nous avons choisi de ne pas stocker le plateau après le coup en mémoire directement car cette représentation nous paraissait plus intuitive, et qu'après réflexion nous n'avons pas décelé de différence réelles entre les deux méthodes, que ce soit en terme de performance ou de mémoire dans notre implémentation finale.

Nous pourrions gagner en mémoire avec notre méthode en ne stockant le plateau qu'une seule fois et en donnant cette référence à tous ses enfants dans l'arbre des coups, ce qui nous ferait économiser de la mémoire mais qui en pratique n'aurait pas fait de réelle différence car seule une faible partie de l'arbre est en mémoire à un instant donné.

3.4 Arbre des coups

L'arbre des coups nous permettra d'obtenir tous les coups jouables depuis une position jusqu'à une profondeur n.
Nous représenterons 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 associé au noeud
- enfants : tous les noeuds ayant des coups réalisables depuis le coup du noeud courant

En pratique nous ne générerons pas tout l'arbre jusqu'à la profondeur d'évaluation pour des raisons de coût en mémoire.
En effet un arbre avec 64 coups possibles au départ, une profondeur de recherche de 10 et une taille en mémoire de 256bits (taille sous estimée par rapport à la réalité) demanderait $\frac{64!}{(64-10)!}.frac2568 = 1,7 \times 10^{19}$ octets de mémoire, soit plus de 10 éxaoctets,ce qui est impossible à stocker, que ce soit en mémoire vive ou non.
Nous allons donc générer les coups lorsque nous en allons en avoir besoin et tirer avantage de la faible durée de vie d'une variable locale, ainsi que de l'ordre d'évaluation des coups afin d'avoir seulement un faible nombre de coups chargé en mémoire à un instant donné.

4 Choix des coups

4.1 Negamax

Afin de choisir parmi les différents coups, la méthode la plus populaire (et la plus intuitive) est les Minimax.
L'idée est que nous allons toujours chercher à maximiser notre score en choisissant un coup et que notre adversaire va chercher à minimiser notre score (maximiser le sien). Cela est exploitable car le gomoku est un jeu à somme nulle et ainsi chaque joueur à tout intéet à gagner.
En générant tous les coups jusqu'à une profondeur choisie nous allons successivement vouloir soit maximiser soit minimiser notre score. En pratique nous allons appeler récursivement la procédure sur tous les coups enfants jusqu'à ce que nous atteignons la fin de l'arbre, et là seulement nous utiliserons notre fonction d'évaluation de position et le score trouvé sera remonté jusqu'à notre coup actuel afin de décider 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 nécessite que la fonction d'évaluation renvoie un résultat relatif au joueur, tel que score(Joueur1) = -score(Joueur2).
L'algorithme est simplifié 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 ← −∞
6:      for all child in node.children do
7:          score ← −NEGAMAX(child, depth − 1)
8:          if score > max then
9:              max ← 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 à choisir, nous allons donc utiliser une variante RootNegamax qui va être quasi identique sauf que juste après la ligne 9 nous allons aussi sauvegarder le coup. Cette méthode nous permet de sauvegarder un coup uniquement si il est jouable directement et de ne pas sauvegarder par mégarde un coup plus bas dans l'arborescence.

Cet algorithme fonctionne très bien mais nous devons quand même évaluer un nombre de positions grandissant énormément en fonction de la profondeur :
nombre de noeuds = $\frac{\text{(nombre de coups possibles initialement)!}}{\text{(nombre de coups possibles initialement - profondeur)!}}$
Heureusement nous avons une méthode très efficace pour réduire le nombre de coups évalués : l'élagage Alpha Beta.

4.2 Élagage alpha-beta

L'élagage alpha beta est un algorithme très puissant qui dans le meilleur des cas va évaluer $\sqrt{n}$ noeuds contre n pour le négamax et n dans le pire des cas.
Le principe est le suivant : Un noeud qui ne peux pas contribuer positivement à l'évaluation n'a pas à être évalué.
En effet lorsque nous sommes surs que continuer à évaluer ne changera pas le résultat, autant s'arrêter.
Cela va êter réalisé grace à deux variables α et β qui vont former l'intervalle des valeurs qui pourraient contribuer à l'évaluation.

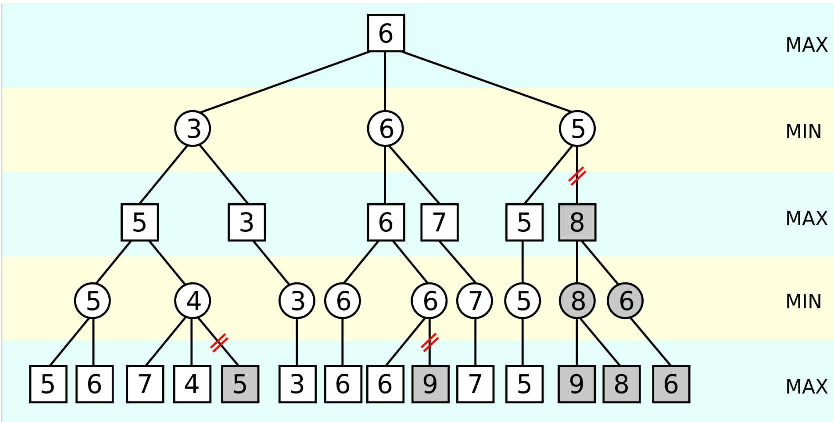

Figure 3 – Exemple d'éxécution de l'algorithme alpha beta. Source : `https://commons.wikimedia.org/wiki/File:AB_pruning.svg`

Dans cet exemple nous avons trois élagages qui ont eu lieu (tous les noeuds sont affichés évalués pour la clarté), de gauche à droite :
Nous cherchons à maximiser la valeur du noeud 5 à la profondeur 3 mais à minimiser la valeur de ses enfants. Ainsi pour que le noeud 4 soit utile il faudrait que sa valeur puisse encore dépasser 5 (notre valeur d'α) en explorant ses enfants. Or comme nous minimisons sa valeur nous ne pourrons jamais dépasser 4, et par conséquent, ainsi il est inutile de continuer et d'valuer le noeud qui aurait valu 5. Nous avons économisé une évaluation ici.
Nous cherchons ensuite à maximiser la valeur du noeud 6 à la profondeur 3 en minimisant la valeur de ses enfants. Ainsi le second noeud 6 n'est intéressant que si il peut dépasser le 6 (notre valeur d'α) déjà calculé, or c'est impossible pour la même raison que précédemment, donc nous gagnons encore une évaluation.
Enfin, nous cherchons à minimiser le noeud 5 à la profondeur 2 tout en maximisant ses enfants. Nous avons déterminé que sa valeur ne pouvais pas être supérieure à 5, or pour être utile il faudrait qu'elle puisse dépasser 6 (notre α) pour remplacer la valeur du noeud 6 de profondeur 1. Ainsi il est inutile de regarder le reste de sons sous arbre car cette partie est inutile. Nous gagnerons ici 6 évaluations.
Dans ce cas nous aurions évalué 33 noeuds avec le négamax et seulement 25 avec l'alpha beta, ce qui représente un gain non négligeable. Nous sommes loin de la valeur optimale théorique mais l'amélioration est conséquente et le sera de plus en plus lorsque la taille de l'arbre augmentera.

Dans la même idée que précédemment pour passer du Minimax au Négamax nous pu adapter la procédure pour qu'elle soit correcte dans notre framework négamax. L'algorithme est plus court mais pas forcément plus simple que la version reposant sur le minimax. Il est le suivant :

Algorithm 3 Algorithme de l'élagage 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 ← −AlphaBeta(child, −beta, −alpha, depth − 1)
7:          if score >= beta then
8:              return beta
9:          end if
10:         if score > alpha then
11:             alpha ← score
12:         end if
13:     end for
14:     return alpha
15: end procedure
```

Comme pour le Negamax nous appelons initialement cette procédure depuis une version modifée qui récupérera la coup après la ligne 11.
Les paramètres α et β seront respectivement initialisés à $-\infty$ et $+\infty$.
Une façon de réduire le nombre de noeuds évalués serait de considérer à la gauche de l'arbre (les coups évalués en premiers) les coups potentiellement bons en choisissant une heuristique telle que la distance au centre du pion posé par le coup, car un coup a plus ce chance d'être bon si il est au centre.

5 Évaluation d'une position

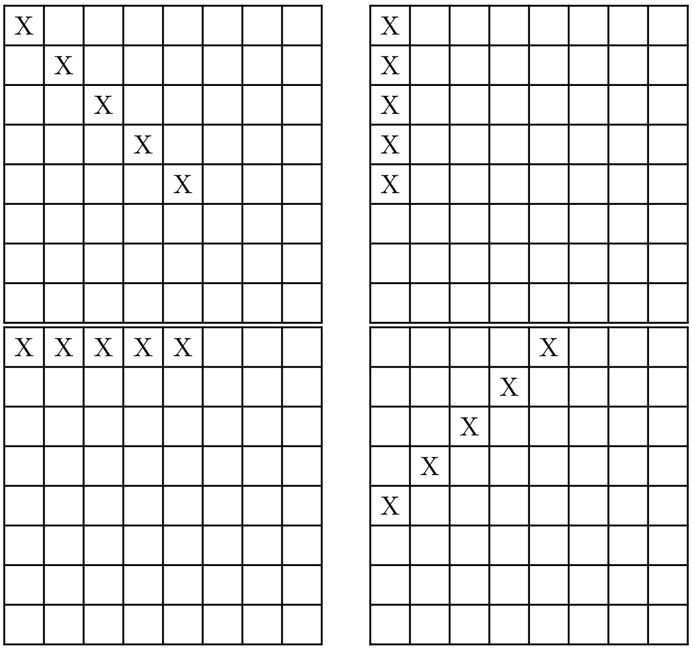

Figure 4 – Patterns vérifiés pour la victoire ou non

6 Conlusion

6.1 Résultats

6.2 Pour aller plus loin