Rapport de projet : Gomoku

Quentin Garrido, Antoine Gélin, Kévin Lor

22 février 2019

Table des matières

1	Introduction	2
2	Utilisation	2
3	Structures de données 3.1 Représentation du plateau	4 5
4	Choix des coups 4.1 Negamax 4.2 Élagage alpha-beta	6 6
5	Évaluation d'une position	7
6	Conlusion 6.1 Résultats	

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 8x8 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://qithub.com/qarridoq/qomoku.

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 8x8:

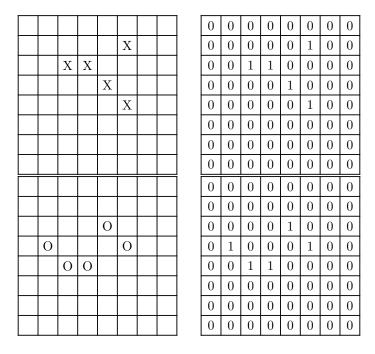


FIGURE 1 – Représentation d'un plateau avec les bitboards associés

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.

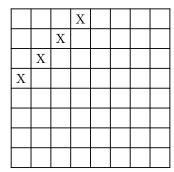
C'est ce choix de représentation qui a motivé notre choix d'utiliser le C++ pour le projet. En effet il est un des rares langages qui nous permet d'accéder à chaque bits individuellement d'un nombre et de réaliser des opérations logiques bit par bit.

Pour représenter un plateau de $n \times n$ il nous faut n^2 bits d'où le choix du plateau 8×8 car un entier au delà de 64 bits 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.



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

```
1: procedure PATTERN_MATCHING(bitboard, pattern)
 2:
       count \leftarrow 0
       for i \leftarrow 0 to 8 – pattern.height do
 3:
            for j \leftarrow 0 to 8 – pattern.width do
 4:
 5:
               if pattern & bitboard = pattern then
                                                                      ▷ & représente un ET logique bit par bit
                   count \leftarrow count + 1
 6:
               end if
 7:
                                                                        \triangleright >> est un décalage de n bits à droite
 8:
               pattern \leftarrow pattern >> 1
 9:
            pattern \leftarrow pattern >> pattern.width-1
10:
       end for
11:
12:
       return count
13: 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 $\frac{64!}{(64-10)!} \cdot frac2568 = 1,7 \times 10^{19}$ 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

Algorithm 2 Algorithme du Negamax

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

4.2 Élagage alpha-beta

5 Évaluation d'une position

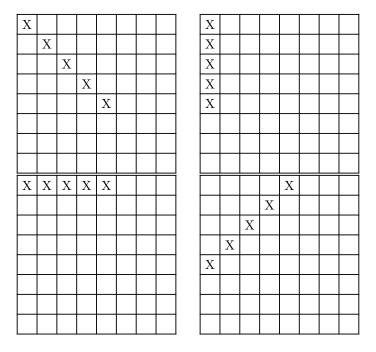


FIGURE 3 – Patterns vérifiés pour la victoire ou non

- 6 Conlusion
- 6.1 Résultats
- 6.2 Pour aller plus loin