

# Projet IN104 (2025) - Programmation d'une IA pour le jeu d'échecs en C

Encadrant : Tom Boumba

Fiche 1 : Bitboards

## Motivation : l'algorithme Minimax

Lorsqu'un joueur souhaite déterminer quel est le meilleur coup qu'il puisse jouer dans l'état actuel de l'échiquier (on parle de "position de l'échiquier"), il peut considérer le résultat de chacun de ses coups possibles. Pour chacun de ces coups, il peut alors considérer l'ensemble des coups jouables dont disposerait son adversaire au tour suivant, et l'ensemble des réponses qu'il pourrait avoir à chacun de ces coups adverses, et ainsi de suite.

Le joueur effectue ainsi un parcours en profondeur du sous-arbre du jeu d'échecs dont les noeuds sont les positions considérées, le noeud racine étant la position actuelle, et les arcs sont les coups occasionnant une transition d'une position de l'échiquier à la position du tour suivant.

Après avoir exploré l'arbre des possibilités avec la plus grande profondeur possible, le joueur peut alors évaluer les positions correspondant aux noeuds feuille de cet arbre. Pour un noeud feuille donné, la valeur est plutôt positive si les blancs ont l'avantage, et plutôt négative si les noirs ont l'avantage. Si un noeud feuille est terminal (i.e un noeud dans lequel la partie de jeu est terminée, par exemple une position de l'échiquier dans laquelle l'un des rois est échec et mat), on peut lui attribuer une valeur exacte (par exemple  $+\infty$  si le roi noir est échec et mat). Si c'est un noeud interne de l'arbre, on doit estimer la valeur du noeud selon une heuristique élaborée "à la main", ou par un réseau de neurones entraîné sur une base de données de parties jouées à haut niveau.

En partant des valeurs des noeuds feuille à une certaine profondeur donnée, on peut alors estimer le meilleur coup possible à jouer dans la position actuelle du noeud racine en remontant l'arbre de noeud en noeud, chaque noeud transmettant à son parent la meilleure valeur de ses noeuds fils selon la couleur du joueur associé : un noeud où le trait est aux blancs (i.e, où c'est aux blancs de jouer) transmet à son parent la plus haute valeur de ses noeuds fils, tandis qu'un noeud où le trait est aux noirs transmet la plus basse valeur de ses noeuds fils. Cette méthode de recherche

du meilleur coup possible est connue sous le nom d'algorithme "Minimax", nommé ainsi car il est caractérisé par l'alternance de couches minimisantes et maximisantes lors de la remontée des valeurs des noeuds feuille vers le noeud racine.

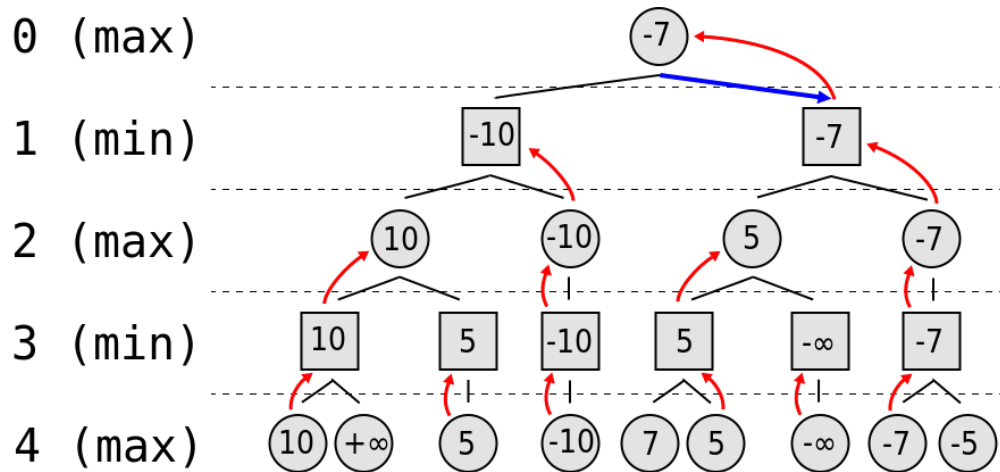


Figure 1: Exemple d'exécution de l'algorithme Minimax

## Programmation du jeu d'échecs et bitboards

Un moteur d'échecs fonctionnant selon l'algorithme Minimax doit pouvoir représenter en mémoire les positions du jeu d'échecs. Il doit aussi pouvoir générer l'ensemble des coups possibles (dits coups "légaux") dans chaque position pour parcourir l'arbre du jeu, ainsi qu'évaluer les positions correspondant aux noeuds feuille de l'arbre.

Plus le nombre de noeuds parcourus, ainsi que la profondeur de parcours de l'arbre des coups possibles sont élevés, plus l'algorithme Minimax est performant. Ces deux facteurs sont limités par la capacité de stockage de l'arbre des positions, ainsi que par la rapidité de la génération des coups possibles dans chaque position et de l'évaluation des positions des noeuds feuille.

Intuitivement, on représente souvent un échiquier à l'aide d'un tableau de 64 caractères prenant 8 bits chacun en mémoire :

```
char echiquier[64];
```

Un échiquier prend alors  $64 \times 8 = 512$  bits en mémoire. Il y a plus efficace : une représentation de l'échiquier par "bitboards". Un bitboard est un entier de 64 bits dont les bits représentent la présence ou non d'une information donnée sur une case de l'échiquier :

```
unsigned long long bitboard;
```

Un bitboard peut représenter l'occupation d'un type de pièce sur l'échiquier (voir figure 2).

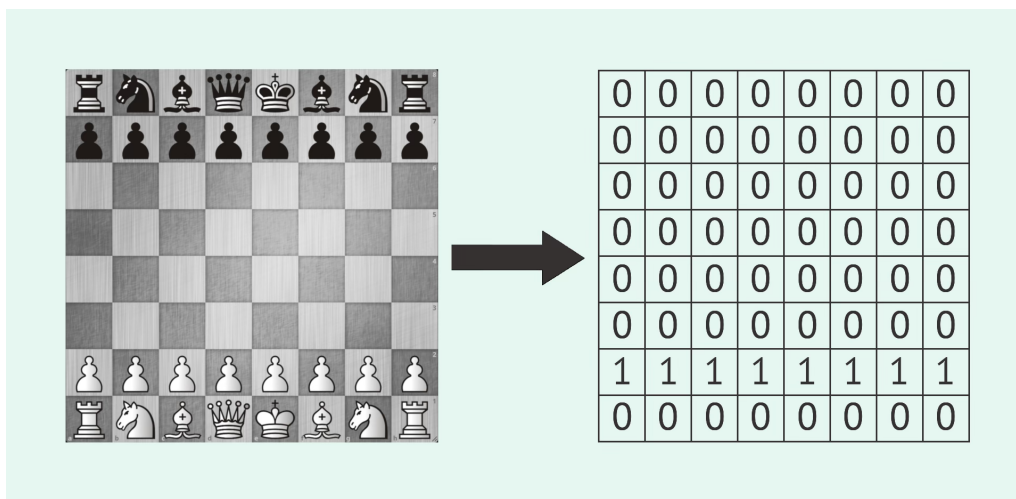


Figure 2: Bitboard d'occupation des pions blancs sur l'échiquier

En différenciant selon la couleur, il existe 12 types de pièce différents sur l'échiquier. Toute position de l'échiquier est donc entièrement représentable par un ensemble de 12 bitboards d'occupation, prenant  $12 \times 8 = 96$  bits en mémoire.

Outre l'avantage en termes de capacité de stockage, la représentation par bitboards permet d'effectuer efficacement toutes les opérations nécessaires pour générer l'ensemble des coups légaux dans une position donnée à l'aide d'opérations binaires. On utilise par exemple la notion de "masque d'attaque" pour représenter l'ensemble des mouvements possibles d'une pièce sur l'échiquier (voir figure 3).

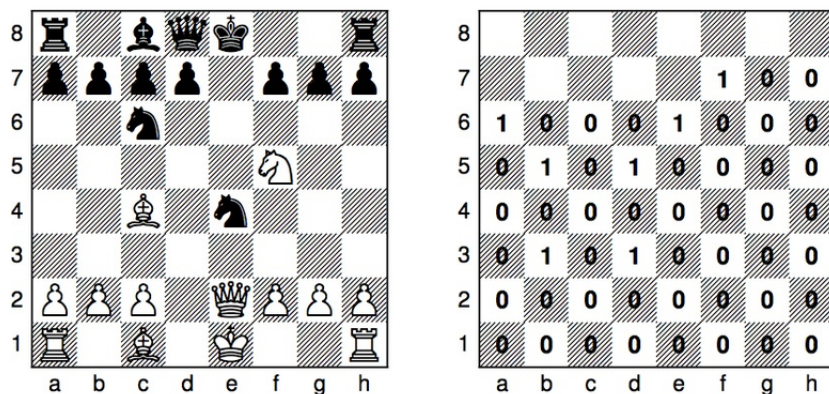


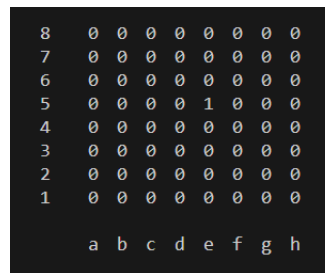
Figure 3: "Masque d'attaque" du fou blanc en C4 : bitboard dont les bits activés représentent la possibilité d'attaque du fou blanc sur l'échiquier. On peut simuler la disparition du pion noir de l'échiquier en F7 lors de sa prise du pion noir par le fou blanc en effectuant une opération binaire 'ET' entre le bitboard d'occupation des pions noirs, et un bitboard contenant un 0 en F7 et un 1 sur l'ensemble des autres cases de l'échiquier.

## Implémentation

Dans le fichier `bitboards.h`, on a déclaré la structure de données `BB` représentant un bitboard, ainsi que les macros `get_bit(bb, square)`, `set_bit(bb, square)`, et `pop_bit(bb, square)`.

- Implémenter la fonction `void print_bitboard(BB bb)`, qui sera utile pour le débogage. Le résultat devrait être le suivant (à noter qu'on se réfère aux indices des différentes cases de l'échiquier en utilisant leurs noms au moyen d'un `enum` défini dans le fichier `echiquier.h`) :

```
BB bb = 0ULL;
set_bit(bb, e5);
print_bitboard(bb);
```



8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

Figure 4: Bitboard dont le bit représentant la case e5 est allumé