

ENSEIRB-MATMECA I1  
Rapport de projet d'algorithmique et de programmation n°1  
*MANSUBA*

Abderahim LAGRAOUI

Finn ROTERS

Encadrant : Idris DULAU

13 janvier 2023



---

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Présentation générale du projet	3
1.2	Problématique	3
1.3	Architecture du projet	3
<b>2</b>	<b>Définition du monde</b>	<b>4</b>
2.1	Tableau du jeu	4
2.1.1	Abstraction du monde	4
2.1.2	La structure <i>world_t</i>	4
2.1.3	Initialisation du monde	5
2.2	Relations définies sur le monde	5
2.2.1	La structure <i>neighbors_t</i>	5
2.2.2	Identification des voisins	5
<b>3</b>	<b>Construction du jeu</b>	<b>6</b>
3.1	Définition des structures de données	6
3.1.1	Types structures	6
3.1.2	Types énumérés	7
3.2	Initialisation séparée d'une partie du jeu	8
3.2.1	Initialisation des informations du jeu	8
3.2.2	Initialisation du tableau de jeu	9
3.3	Déplacements	9
3.3.1	Mouvement simple	9
3.3.2	Saut simple	10
3.3.3	Sauts multiples	10
3.3.4	Mouvement d'un éléphant	11
3.3.5	Mouvement d'une tour	11
3.4	Mise à jour du tableau de jeu	11
3.5	Détermination de la fin d'une partie	12
3.6	Génération d'une partie aléatoire	13
3.7	Affichage du jeu	13
<b>4</b>	<b>Autres tableaux de jeu</b>	<b>14</b>
4.1	La grille triangulaire	14
4.1.1	Construction	14
4.1.2	Relations	14
4.1.3	Déplacements	15
4.1.4	Initialisation du tableau de jeu	16
4.1.5	Affichage du tableau	16
4.2	La grille d'échecs classiques	17
4.2.1	Construction	17
4.2.2	Relations	17
4.2.3	Déplacements	18
4.2.4	Initialisation du tableau de jeu	19
4.2.5	Affichage du tableau	19
<b>5</b>	<b>Améliorations possibles</b>	<b>20</b>
5.1	Stratégies gagnantes	20
5.2	Interface Homme-Machine	20
5.3	Performance du code	20
<b>6</b>	<b>Conclusion</b>	<b>21</b>

---

# 1 Introduction

## 1.1 Présentation générale du projet

Étant introduit au monde européen par les arabes au X<sup>e</sup> siècle, le jeu d'échecs est l'un des jeux de réflexion les plus populaires au monde. Le projet du module "*Projet d'algorithmique et de programmation N° 1*" s'intéresse particulièrement à l'une des anciennes formes de ce jeu c'est "Manşūba". Il s'agit d'un jeu précurseur du jeu de société chinois des dames. Dans notre modèle simplifié, une partie se joue à deux, les deux joueurs occupent initialement des positions symétriques dans le tableau du jeu, chaque pièce selon son type a plusieurs possibilités de déplacement. Le critère d'une victoire, qui se fixe en début de partie, se présente en deux variétés : une victoire simple ou une victoire complexe. L'objectif de ce projet est de s'intéresser à ces jeux de plateaux avec des pièces qui se déplacent de manière plus ou moins loufoques, dans des tableaux de de jeu variés.



FIGURE 1 – Tableau de jeu de dames chinoises

## 1.2 Problématique

La problématique principale était de comprendre et de faire une abstraction du jeu en appliquant toutes nos connaissances sur la programmation en langage C, de bien échanger les idées et les connaissances au sein du groupe et de savoir comment résoudre les différents problèmes sur le niveau compilation mais aussi sur le niveau de manipulation de notre dépôt *Git*. De plus, l'exercice c'était de implémenter un code qui se base sur les structures de données et les fonctions prédéfinies dans la version de base.

## 1.3 Architecture du projet

Le projet est constitué d'un ensemble de fichiers écrits en langage de programmation C. Pour compiler le projet, on utilise l'outil Makefile qui crée un exécutable pour chaque fichier et ensuite compile l'ensemble des fichiers en un seul exécutable appelé *projet*. Afin d'améliorer l'indépendance entre les fichiers, nous avons créé un *header* pour chaque fichier qui contient le minimum des *#include* nécessaires pour la compilation. Pour les tests, nous avons adopté une stratégie qui consiste à effectuer autant de tests que possible pour chaque fonction, en changeant les valeurs des arguments et en traitant des cas extrêmes, comme pour la fonction *get\_neighbors* où nous avons testé des exemples de positions au milieu du tableau ainsi que des positions aux bords du tableau. À propos des options du jeu tels la taille de la grille et le nombre de tours, on a utilisé la fonction *getopt*.

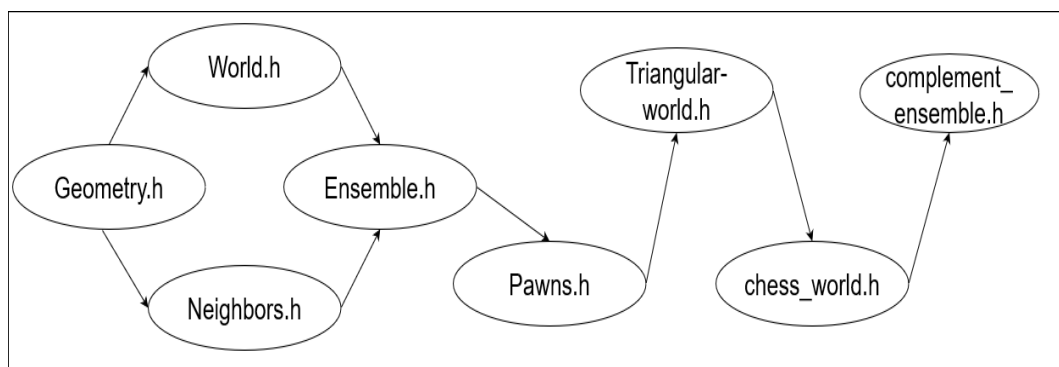


FIGURE 2 – Graphe des dépendances

---

## 2 Définition du monde

### 2.1 Tableau du jeu

#### 2.1.1 Abstraction du monde

La contrainte d'optimisation du code en termes de complexité nous a amené à ne pas implémenter le monde comme étant un tableau 2D, car en effet à chaque parcours d'une position, il faudra au moins deux boucles imbriquées. Alors on a choisi de l'imaginer comme étant un seul tableau de taille *WORLD\_SIZE* et chaque début de ligne est caractérisé par une position *idx* telle que  $idx \equiv 0 \pmod{WIDTH}$ . Voilà un exemple illustrant ce principe sur la figure 3.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

FIGURE 3 – Tableau de jeu 10 x 10

#### 2.1.2 La structure *world\_t*

L'implémentation du type abstrait *world\_t* est désormais évidente grâce au principe expliqué dans la partie précédente. Comme l'illustre la figure 4, cette structure qui représente l'état de toutes les cases, contient deux champs : un tableau *colors* de type énuméré *color\_t* de taille *WORLD\_SIZE* et un autre tableau *sorts* de type énuméré *sort\_t* de même taille.

```
struct world_t{
    enum color_t colors[WORLD_SIZE];
    enum sort_t sorts[WORLD_SIZE];
};
```

FIGURE 4 – La structure *world\_t*

---

### 2.1.3 Initialisation du monde

Avant de commencer une partie, il faut d'abord créer un tableau de jeu vide, pour cela on utilise la fonction *world\_init* qui remet les deux champs d'une variable globale *monde*, définie exclusivement dans le fichier *world.c*, à *NO\_COLOR* et *NO\_SORT*. Dans la première implémentation de cette fonction, on a utilisé la fonction *malloc* pour allouer de l'espace mémoire afin de pouvoir retourner un pointeur vers le tableau de jeu, mais on a eu des erreurs de type *valgrind*. Et à cause de cela, on était obligé de s'en sortir avec la variable globale *monde*. *world\_init* a une complexité linéaire  $O(n)$ .

## 2.2 Relations définies sur le monde

### 2.2.1 La structure *neighbors\_t*

Puisque chaque pièce a une dynamique de mouvement, il faut absolument mettre en évidence les relations qu'elle maintient avec ses voisins. Dans le modèle de base du jeu, selon sa position chaque pièce a un nombre de voisins qui varie entre 3 et 8 voisins et chaque position peut être occupée par au plus une seule pièce.

L'implémentation de ces aspects se fait grâce à deux structures dépendantes des deux figures 5 et 6 : *vector\_t* qui contient un champ *i* de type entier représentant l'indice du voisin et un champ *d* de type énuméré *dir\_t* désignant la direction dans laquelle se situe, et la structure *neighbors\_t* sous forme d'un tableau de type *vector\_t* de taille *MAX\_NEIGHBORS*.

```
struct vector_t {
    unsigned int i; // the index of the place the vector is pointing to
    enum dir_t d;   // the direction towards this place
};
```

FIGURE 5 – La structure *vector\_t*

```
struct neighbors_t {
    struct vector_t n[MAX_NEIGHBORS];
};
```

FIGURE 6 – La structure *neighbors\_t*

### 2.2.2 Identification des voisins

La gestion du tableau de jeu nécessite de plus l'identification de la liste des voisins de chaque case, la fonction *get\_neighbors* s'en sert. Le principe algorithmique de cette fonction est de distinguer entre deux zones critiques du tableau comme s'est montré sur la figure 7, la zone au milieu en gris et la zone constituée par les 4 frontières en jaune. Et puis selon la zone à laquelle appartient la case prise comme argument, on retourne une structure *vector\_t* indiquant la liste de ses voisins. Il est à noter que dans tous les types des grilles qu'on définira, cette fonction retourne une liste de 8 voisins, les voisins avec une valeur *UINT\_MAX* ne seront pas pris en compte.

La complexité de la fonction *get\_neighbors* est en  $O(n)$ , avec  $n$  égale à *MAX\_NEIGHBORS*.

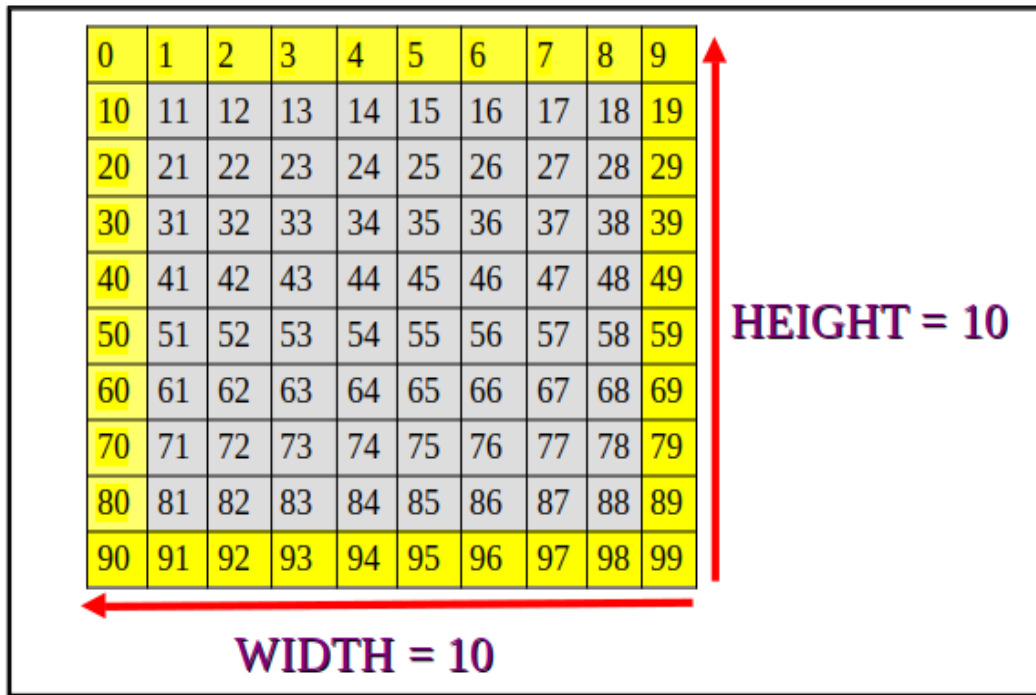


FIGURE 7 – Zones critiques dans un tableau de jeu

### 3 Construction du jeu

#### 3.1 Définition des structures de données

##### 3.1.1 Types structures

Après avoir créer le tableau du jeu, il est désormais possible de mettre en place les règles du jeu. Pour cela on définit une nouvelle structure de données *positions\_info* sur la figure 8, cette structure associe à chacun des deux joueurs, quatre tableaux représentant l'ensemble de ses informations relatives, celui des positions initiales, celui des positions actuelles qu'il occupe et celui de l'état d'emprisonnement de ces positions. De plus, elle contient deux entiers non signés *MAX\_TURNS* et *TURNS* représentant respectivement le nombre maximal de tours possibles et le nombre de tours joués. Pour une contrainte de complexité, le champ des mouvements possibles pour chaque joueur était omis. Cependant ce besoin va être réglé ailleurs .

```
// the allowed moves and the where the pieces are at the moment
struct positions_info {
    /* at this poin we don't have to define a structure for possible mouvements of each player */
    unsigned int initial_WHITE[HEIGHT]; // initial positions of the player with white pawns
    unsigned int current_pieces_WHITE[HEIGHT]; // current positions of the player with white pawns
    unsigned int initial_BLACK[HEIGHT]; // initial positions of he player with black pawns
    unsigned int current_pieces_BLACK[HEIGHT]; //current positions of the player with black pawns
    unsigned int MAX_TURNS; // Maximum allowed turns = WORLD SIZE
    unsigned int TURNS; // Played turns in the game.
};
```

FIGURE 8 – La structure *positions\_info*

---

La structure *move* de la figure 9, sert à discrétiser un déplacement donnée, elle contient les indices de départ et d'arrivée de la pièce concernée ainsi que le type du déplacement. Elle sera très utile dans la génération d'une partie aléatoire.

```
struct move{
    unsigned int ex_idx;
    unsigned int new_idx;
    enum move_types type;
};
```

FIGURE 9 – La structure *move*

### 3.1.2 Types énumérés

Voilà la liste des types énumérés utilisés dans l'implémentation du jeu.

Comme c'est défini sur la figure 10, les couleurs possibles des pièces sont le blanc et le noir.

```
enum color_t {
    NO_COLOR = 0, // Default color, used to initialize worlds
    BLACK = 1,
    WHITE = 2,
    MAX_COLOR = 3, // Total number of different colors
};
```

FIGURE 10 – Le type énuméré *color\_t*

Les directions possibles sont les huit directions mentionnées dans la figure 11.

```
enum dir_t {
    NO_DIR = 0, // Default dir (i.e unset)
    EAST = 1,
    NEAST = 2,
    NORTH = 3,
    NWEST = 4,
    WEST = -1,
    SWEST = -2,
    SOUTH = -3,
    SEAST = -4,
    MAX_DIR = 9, // Total number of different directions
};
```

FIGURE 11 – Le type énuméré *dir\_t*

Comme elle le montre la figure 12, les types des pièces mises en jeu le simple pion *PAWN*, la tour *TOWER*, l'éléphant *ELEPHANT* et le sauteur *BISHOP*.

```

/** Enum defining the possible sorts of pieces in the world */
enum sort_t {
    NO_SORT = 0,    // Default sort (i.e nothing)
    PAWN     = 1,
    TOWER    = 2,
    ELEPHANT = 3,
    BISHOP   = 4,
    MAX_SORT = 5,
    // Total number of different sorts
};

```

FIGURE 12 – Le type énuméré *sort\_t*

Comme elle le montre la figure 13, une victoire peut être simple ou complexe. le type *move\_types* ne sera utilisé que dans la génération de la partie aléatoire dans la version de base du jeu, pour cela on ne considère que les trois types de mouvements définis pour les simples pions.

```

enum move_types{
    SIMPLE_MOVE = 1,
    SIMPLE_JUMP = 2,
    MULTIPLE_JUMP = 3,
};

//we add the types of victory
enum victory{
    COMPLEX_WIN = 1,
    SIMPLE_WIN = 2,
};

```

FIGURE 13 – Les types énumérés *victory* et *move\_types*

## 3.2 Initialisation séparée d'une partie du jeu

Grâce à tous les outils définis précédemment il est possible maintenant de commencer le jeu. À cause des problèmes liés aux tailles des tableaux dans les deux structures *world\_t* et *positions\_infos*, on a décidé d'initialiser séparément le tableau de jeu et les informations du jeu en utilisant deux fonctions indépendantes, et ces dernières vont être utilisés une seule fois au début de la partie .

### 3.2.1 Initialisation des informations du jeu

La fonction *init\_infos* associe aux deux joueurs leurs positions initiales qui seront confondues avec leurs positions actuelles au début de la partie comme le montre la figure 14, on suppose que le joueur avec les pièces blanches occupe initialement toutes les positions de la dernière colonne à gauche du tableau de jeu (en rouge), et l'autre joueur occupe symétriquement celles à droite (en vert).



0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

**HEIGHT = 10**

**WIDTH = 10**

FIGURE 14 – Un tableau de jeu au début d’une partie

### 3.2.2 Initialisation du tableau de jeu

La fonction *init\_players\_0* applique le même principe de la fonction *init\_infos* sauf qu’elle agit sur le monde c’est à dire sur le monde physique définie précédemment .

## 3.3 Déplacements

Les mouvements des joueurs sont indispensables au jeu. On distingue trois types , déplacement simple, un saut simple et des sauts multiples. À son tour, chaque joueur peut effectuer l’un de ces mouvements. Pour le déplacement et saut simples, on prend comme argument la position d’arrivée et on utilise une fonction auxiliaire booléenne, permettant de savoir si ce déplacement est possible. Pourtant pour les sauts multiples, on fait une suite de sauts simples tant qu’il est possible sous réserve de ne pas revenir à la position initiale. Finalement, après chaque déplacement, le monde et les informations du jeu sont mis à jour.

### 3.3.1 Mouvement simple

Avec un déplacement simple, une pièce se déplace vers une position voisine à condition que cette dernière soit libre. On distingue entre les déplacements des pièces noires et des pièces blanches. On autorise uniquement les déplacements vers une position gagnante, cela signifie que les pions blancs partent à l’ouest et ne peuvent se déplacer que dans les directions nord, nord-est, est, sud-est ou sud comme s’est montré sur la figure 15. Noir part de l’est et se déplace dans les directions nord, nord-ouest, ouest, sud-ouest ou sud comme l’illustre la figure 16.

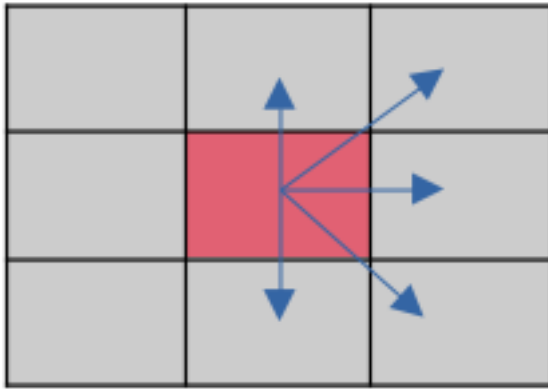


FIGURE 15 – Déplacement d'un blanc

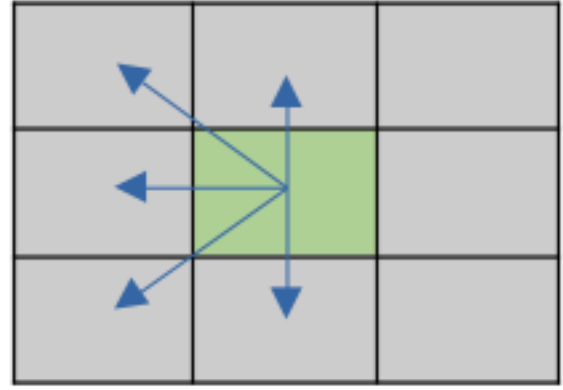


FIGURE 16 – Déplacements d'un noir

### 3.3.2 Saut simple

Lors d'un saut simple, le pion saute par-dessus un autre pion. Ainsi, le pion peut se déplacer plus rapidement qu'avec un déplacement simple. La condition préalable à ce mouvement est que la position sur laquelle le pion saute soit libre et que la position de passage entre les deux positions soit occupée par un pion de n'importe quelle couleur, voir la figure 17.

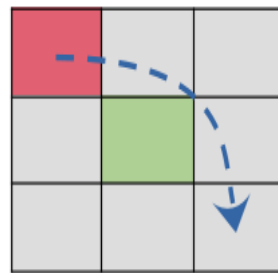


FIGURE 17 – Exemple d'un saut simple

### 3.3.3 Sauts multiples

Un saut multiple est une suite de sauts simples, voir figure 18. Tant que il est possible, on effectue des sauts simples et la fonction qui effectue ce déplacement retourne la position finale du pion. Il est impératif de ne pas revenir à la position initiale, sinon un saut infiniment long et inutile peut être généré.

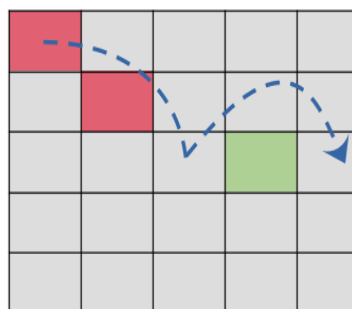


FIGURE 18 – Exemple d'un saut multiple

### 3.3.4 Mouvement d'un éléphant

L'éléphant comme s'est montré sur la figure 19 avec le caractère &, effectue deux mouvements simples dans deux directions cardinales possibles pour le joueur concerné, et cela sans aucune exigence sur la case intermédiaire qu'il soit libre ou non.

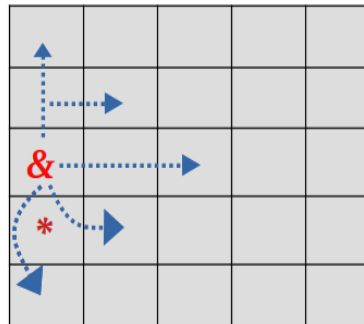


FIGURE 19 – Déplacements d'un éléphant

### 3.3.5 Mouvement d'une tour

La tour peut se translater selon une direction cardinale tant qu'elle n'affronte pas de case occupée comme s'est montré dans la figure 20 avec le caractère #.

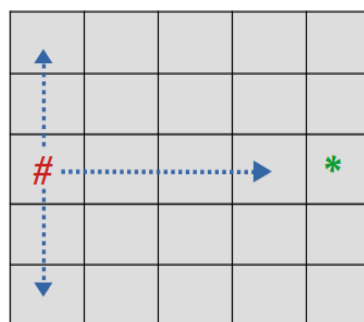


FIGURE 20 – Déplacements d'une tour

## 3.4 Mise à jour du tableau de jeu

Comme c'est déjà mentionné pour les déplacements, le tableau du jeu doit être actualisé après chaque tour. Cela se fait en mettant à jour le tableau du jeu de type *world\_t* et les informations du jeu de type *position\_info*. Pour cela, nous utilisons la fonction *update\_currentpieces*. Après chaque tour, cette fonction réorganise le tableau des positions actuelles des joueurs en remplaçant l'ancienne position par la nouvelle position et change l'état de l'ancienne position en position libre.

Le cas d'une capture simple, c'est lorsque le déplacement d'une pièce se termine sur une case contenant une pièce d'une couleur différente. Dans ce cas, cette dernière pièce sera considérée comme prisonnière. La mise à jour du tableau prend en compte cet effet et si les conditions d'un emprisonnement sont vérifiées on l'effectue en modifiant le champ *status\_pieces* dans les informations du jeu.

---

### 3.5 Détermination de la fin d'une partie

Une partie est terminée lorsqu'un joueur obtient une victoire simple ou une victoire complexe.

#### Victoire simple

Une simple victoire est réalisée lorsqu'un pion appartenant à un joueur atteint une des positions initiales de l'adversaire, un exemple est fourni sur la figure 22. Pour cela, on vérifie s'il existe une position dans un tableau de positions initiales d'un joueur occupée par une pièce appartenant à son adversaire en utilisant les champs de la structure *positions\_info*. La fonction *simple\_win* qui détermine la fin de cette partie a une complexité en  $O(n)$ , où  $n$  est la valeur de *WORLD\_SIZE*.

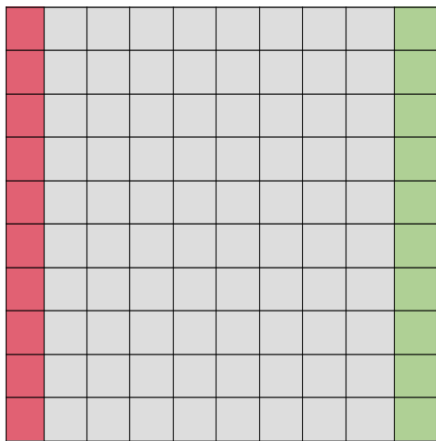


FIGURE 21 – L'état initial du plateau

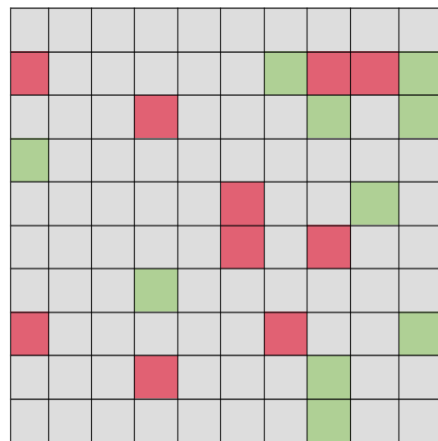


FIGURE 22 – Victoire simple du noir

#### Victoire complexe

Une victoire complexe est réalisée lorsque tous les pions d'un joueur occupent toutes les positions initiales de son adversaire, un exemple est fourni sur la figure 24. La fonction *complex\_win* qui détermine la fin de cette partie a une complexité en  $O(n)$ , où  $n$  est la valeur de *WORLD\_SIZE*.

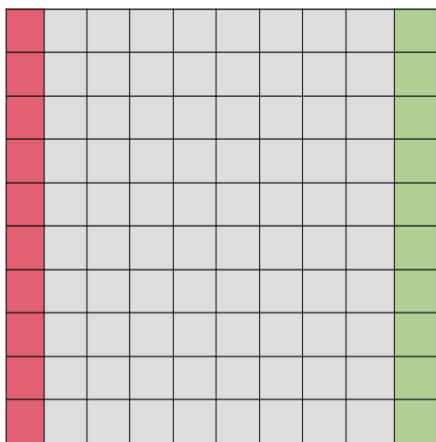


FIGURE 23 – L'état initial du plateau

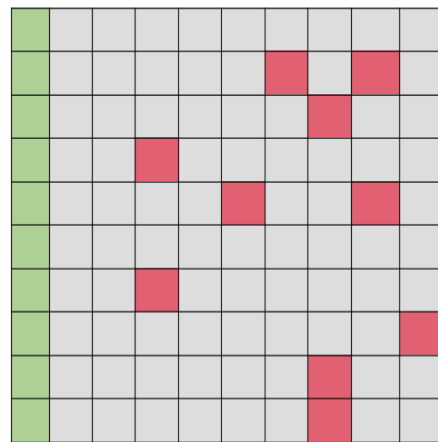


FIGURE 24 – Victoire complexe du noir

---

### 3.6 Génération d'une partie aléatoire

Pour commencer une partie du jeu, des paramètres aléatoires sont générés. Par exemple, un joueur aléatoire est choisi pour démarrer le jeu. Un pion aléatoire du joueur qui démarre est sélectionné, lui même effectue un mouvement possible aléatoire. Il est à noter que dans cette partie, seuls les simples pions sont mis sur le plateau de jeu.

#### Choix aléatoire d'un joueur

Comme il n'y a que deux joueurs dans ce jeu, il faut générer soit un nombre valant 1 pour le joueur avec les pions noirs et 2 pour l'autre joueur. Pour cela, on génère un entier  $r$  avec la fonction `rand()` et comme  $r$  est aléatoire, on retient  $r \bmod(2) + 1$ .

#### Choix aléatoire d'une pièce appartenant à ce dernier

La fonction `choose_random_piece_belonging_to` choisit une pièce aléatoire appartenant au joueur pris comme argument. Pour cela, un nombre aléatoire quelconque est également généré et réduit avec la technique de `mod (HEIGHT)`. Ce nombre représente l'indice dans le tableau `current_pieces` de la pièce à choisir.

#### Choix aléatoire d'un mouvement possible pour cette pièce

La fonction `choose_random_move_for_piece` retourne un `struct move` représentant un déplacement possible pour la pièce choisie. On choisit une position du tableau de jeu, et tant que aucun mouvement vers cette position n'est possible, on prend une autre position aléatoire et on vérifie si un déplacement possible parmi les trois types de déplacements vers cette position est possible. Il est à signaler qu'on favorise dans cet ordre le saut multiple puis le saut simple et enfin le déplacement simple.

#### Exécution du mouvement et mise à jour du monde

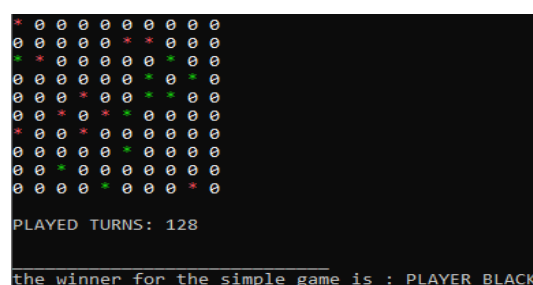
En se basant sur le mouvement aléatoire choisi, on effectue les mises à jour nécessaires du tableau et les informations du jeu en appelant les fonctions déjà implémentées.

#### Passage du tour à l'adversaire

La tour des rôles est maintenue par la fonction `next_player` retourne un `enum players` représentant l'adversaire.

### 3.7 Affichage du jeu

La communication avec les joueurs reste un point primordial dans le jeu, pour cela on utilise la fonction `print_world` qui affiche la grille du jeu après chaque déplacement, on utilise la couleur `rouge` pour les pièces blanches et la couleur `verte` pour celles en noir. De plus les pions sont affichés sous forme du caractère `*` et les positions libres en un `0` blanc. À la fin d'une partie, on affiche le gagnant. Un exemple est fourni sur la figure 25.



```
* 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 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 0 0
0 0 0 0 * 0 0 0 0
PLAYED TURNS: 128
the winner for the simple game is : PLAYER_BLACK
```

FIGURE 25 – Une victoire réalisée par le joueur en pions noirs

---

## 4 Autres tableaux de jeu

### 4.1 La grille triangulaire

#### 4.1.1 Construction

En se basant sur la même définition du tableau de la grille classique, on construit une nouvelle grille appelée grille triangulaire. Ce nouveau plateau de jeu est caractérisé par un nombre de voisins limité à 6 et par deux directions cardinales éliminées. On commence par rendre des positions non jouables tel que une position  $idx$  est non jouable si et seulement si sa couleur est égale à  $MAX\_COLOR$  et le type de sort dedans est de type  $MAX\_SORT$ . Le tableau contient  $HEIGHT$  lignes  $L_0, L_1, L_2, \dots, L_{HEIGHT-1}$ , dans chaque ligne  $L_i$  : si  $i$  est pair on rend la première position de  $L_i$  non jouable (en blanc) et la deuxième jouable (en gris) ainsi de suite comme l'illustre la figure 26, sinon on fait l'inverse.

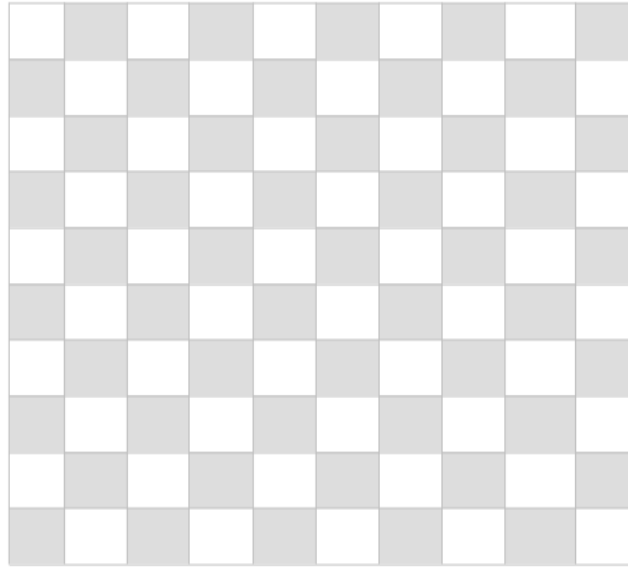


FIGURE 26 – Une grille triangulaire

#### 4.1.2 Relations

Dans la grille triangulaire, chaque position jouable peut avoir au maximum 6 voisins. Pour les directions non cardinales, les voisins restent identiques à ceux de la grille classique. Cependant pour les directions cardinales, on élimine les deux directions *EAST* et *WEST* c'est à dire qu'on ne les considère pas dans la liste des voisins. Et pour les directions *SOUTH* et *NORTH*, on saute la première case dans une direction qui est non jouable et on prend la deuxième case dans la même direction comme s'est montré dans la figure 27.

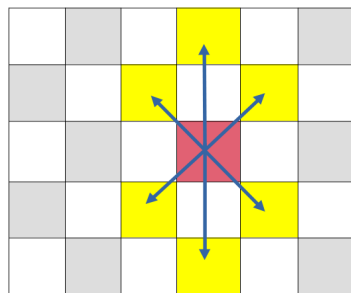


FIGURE 27 – Exemple de repérage de voisins

---

### 4.1.3 Déplacements

Vu l'élimination des deux directions *EAST* et *WEST*, les déplacements des types *ELEPHANT* et *TOWER* sont inutiles. Ainsi dans cette grille, on ne considère que les déplacements des types *PAWN* et du type *BISHOP*. Le *BISHOP* ou le sauteur peut faire des déplacements uniquement selon les directions diagonales possibles pour chaque joueur : dans les directions *SEAST* et *NEAST* pour le joueur en pièces blanches et dans les directions *SWEST* et *NWEST* pour le joueur en pièces noires.

1. Déplacement simple :

Dans ce plateau de jeu, les déplacements simples qui concernant les deux types de pièces existantes sont définies tels que dans le plateau classique : une pièce se déplace vers une position voisine sous réserve qu'elle soit libre et bien évidemment jouable comme c'est montré dans la figure 28. De plus, on favorise pour chaque joueur les déplacements qui le rapproche d'une victoire.

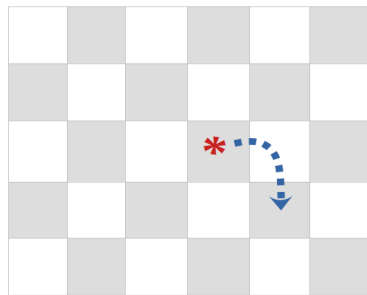


FIGURE 28 – Exemple d'un déplacement simple d'une pièce blanche

2. Saut simple :

Un saut simple est une suite de deux déplacements simples dans la même direction (voir la figure 29).

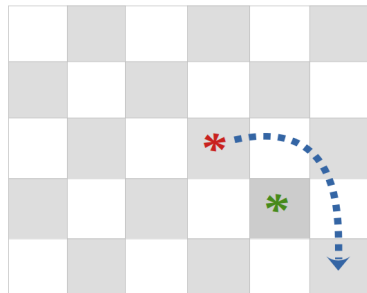


FIGURE 29 – Exemple d'un saut simple d'une pièce blanche

3. Déplacement du SAUTEUR :

Comme c'est expliqué précédemment le *BISHOP* peut se déplacer dans deux directions diagonales possibles pour chaque joueur comme c'est montré sur la figure 30. Cependant, il n'a pas le droit de sauter au dessus d'une autre pièce.

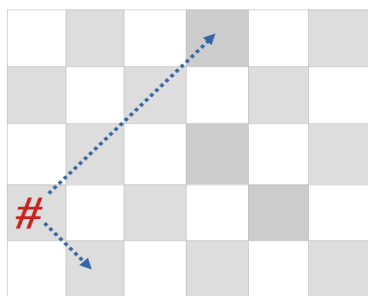


FIGURE 30 – Exemple d'un déplacement d'un sauteur de couleur blanche

#### 4.1.4 Initialisation du tableau de jeu

Pour déterminer les positions de départ des joueurs, on va utiliser une configuration symétrique. Le joueur avec les pièces blanches occupera les positions jouables de la dernière colonne à gauche et remplira ces positions avec des pions simples, à l'exception des première et dernière positions jouables de cette colonne, où il placera des *BISHOP*. Le joueur avec les pièces noires effectuera le même principe dans le côté droit de la grille. C'est ce qui est illustré par la figure 31.

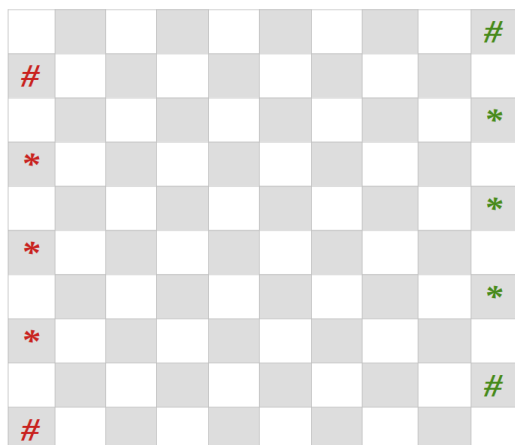


FIGURE 31 – Tableau de jeu initial d'une grille triangulaire

#### 4.1.5 Affichage du tableau

Pour afficher la grille triangulaire, on utilise les mêmes principes d'affichage de la grille classique sauf qu'on ignore les positions non jouables du tableau. Et pour le nouveau type de pièce *BISHOP*, on utilise comme identifiant le caractère '#' tel qu'il est montré dans la figure 32.



FIGURE 32 – Affichage d'une grille triangulaire



---

## 4.2 La grille d'échecs classiques

### 4.2.1 Construction

La grille d'échecs classiques ou l'échiquier, est un tableau de jeu dont les positions sont en alternances sombres et claires. Et chaque pièce ne peut se déplacer que dans les positions ayant la même couleur qu'elle. On construit cette grille de la même manière que la grille triangulaire sauf que les positions non jouables se transformeront en des positions claires et les positions jouables vont être des positions sombres, la figure 33 illustre ce principe. Cependant, il n'est pas possible de définir une position claire comme étant de type *MAX\_SORT*, car cela entraînerait une contradiction avec le fait qu'une position peut être à la fois sombre et occupée. Pour cette raison, le caractère sombre ou clair d'une position n'a pas de signification physique dans notre code. Ainsi pour régler ce problème, la fonction qui identifie les voisins d'une case dans cette grille, prend en compte cette contrainte. Il est à noter que les deux joueurs ne se croisent jamais dans cette grille, donc le statut d'emprisonnement d'une case n'a plus de sens.

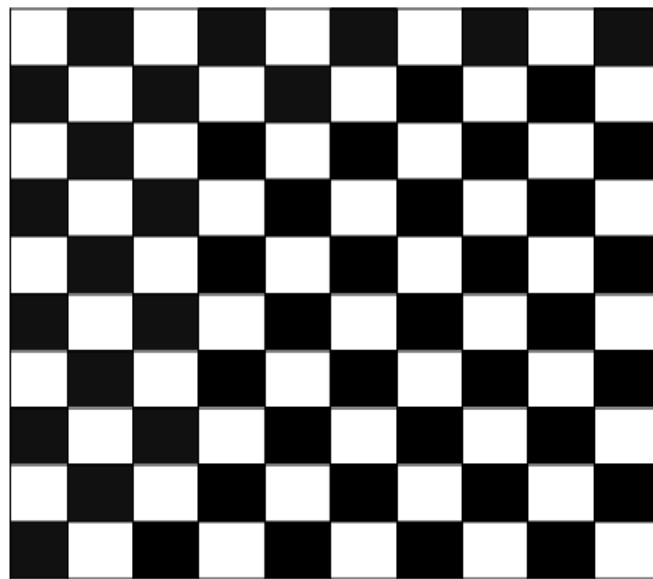


FIGURE 33 – Un échiquier de taille 10x10

### 4.2.2 Relations

Dans cette grille, chaque position peut avoir au plus 4 voisins. Ce sont les positions voisines dans les 4 directions non cardinales comme le montre la figure 34.

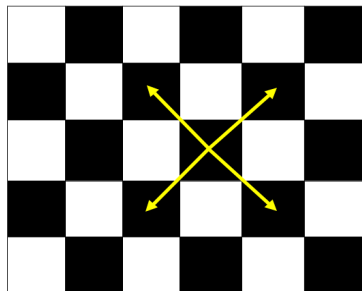


FIGURE 34 – Exemple de voisins dans un échiquier

### 4.2.3 Déplacements

1. Déplacement simple :

Les pions peuvent se déplacer d'une case dans une des deux directions diagonalement opposées selon leur couleur. Les pions blancs peuvent se déplacer vers le nord-est ou le sud-est, tandis que les pions noirs peuvent se déplacer vers le nord-ouest ou le sud-ouest. C'est ce que la figure 35 illustre.

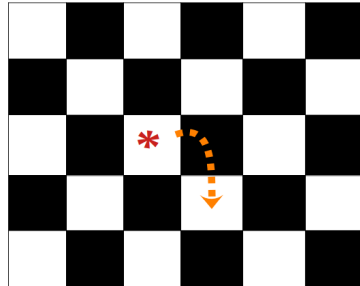


FIGURE 35 – Exemple d'un déplacement simple

2. Saut simple :

La figure 36 montre le principe d'un simple saut sur cette grille. Comme les deux joueurs ne se croisent jamais sur le tableau du jeu, un saut simple avec un pion n'est possible que si la case intermédiaire est occupée par un pion de la même couleur.

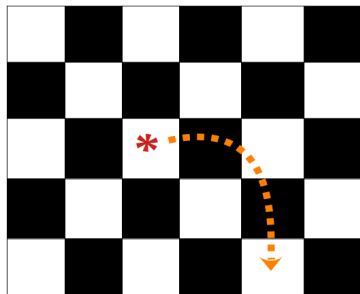


FIGURE 36 – Exemple d'un saut simple

3. Sauts multiples :

Pour les sauts multiples sur un échiquier classique, les mêmes conditions que précédemment s'appliquent (voir section 4.3.3). Comme pour le saut simple, seules les pièces de la même couleur peuvent sauter les unes au dessus des autres. La figure 37 l'illustre.

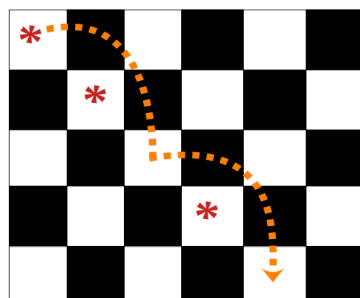


FIGURE 37 – Exemple de sauts multiples

#### 4. Déplacement du SAUTEUR :

Le sauteur conserve les mêmes chemins de déplacements que ceux de la grille triangulaire. En fonction du joueur à qu'il appartient, il peut se déplacer dans deux directions diagonalement opposées. La figure 38 donne un exemple de ce type de déplacement.

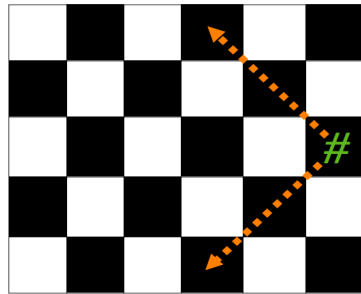


FIGURE 38 – Exemple d'un déplacement avec le sauteur

#### 4.2.4 Initialisation du tableau de jeu

Pour le tableau initial, on utilise la même configuration symétrique que celle de la grille triangulaire avec une translation des positions initiales des pièces noires par une seule case vers le haut. Cette configuration c'est celle présentée dans la figure 39.

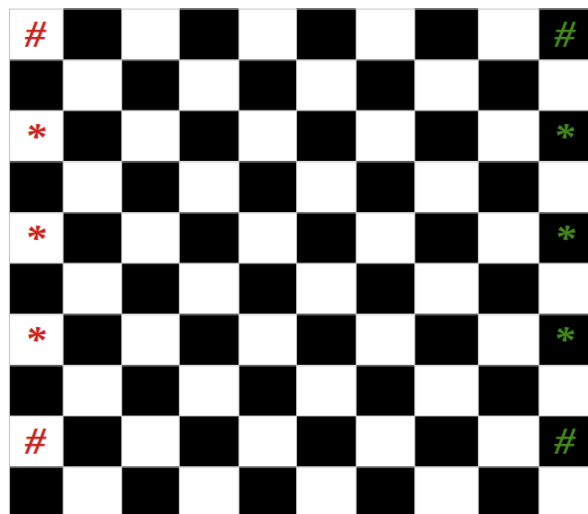


FIGURE 39 – Tableau initial

#### 4.2.5 Affichage du tableau

Cette grille présente la particularité d'avoir des cases sombres ou claires ce qui rend l'affichage difficile. Pour répondre à cet contrainte, on utilise les mêmes principes d'affichage précédents sauf que dans la commande \033, on ajoute un argument qui paramètre la couleur de l'arrière plan. Voilà un exemple sur la figure 40 .



FIGURE 40 – Affichage d'une grille d'échecs classiques

## 5 Améliorations possibles

La totalité de nos travaux dans le projet "*Mansuba*" ne peuvent pas s'arrêter à ce niveau, plusieurs fonctionnalités parmi celles qu'on a pu faire, peuvent être largement améliorées et plusieurs nouvelles idées peuvent perfectionner notre travail. Dans cette partie on parlera de quelques unes, et on fera une analyse critique de notre travail tout au long le projet.

### 5.1 Stratégies gagnantes

Le jeu repose sur des mouvements aléatoires, cependant nous avons veillé à ce que les pions ne soient pas limités à des mouvements répétitifs ou à un mouvement arrière. Nous avons également pris soin de ne pas inclure des stratégies gagnantes trop évidentes. Il aurait cependant été intéressant d'implémenter des options de positionnement des pions qui permettent d'atteindre l'objectif plus rapidement en effectuant des sauts multiples, considérés comme les mouvements les plus puissants. Par exemple, on pourrait déterminer l'ensemble des mouvements possibles pour chaque pièce et les trier suivant un ordre croissant du nombre des cases en avant qu'elles apportent.

### 5.2 Interface Homme-Machine

De plus, il aurait été intéressant de pouvoir jouer contre l'ordinateur en demandant à l'utilisateur une liste de valeurs qui seront pris par la fonction *scanf*, cette fonction permet la saisie de commandes par un utilisateur sur le terminal. Cela aurait non seulement ajouté une dimension de défi supplémentaire au jeu, mais aurait également permis de tester les stratégies proposées dans la première proposition d'amélioration en les utilisant pour essayer de battre l'ordinateur.

### 5.3 Performance du code

Le projet en cours ne priorisait pas la complexité, mais il est important de considérer cet aspect dans d'autres projets ou dans sa carrière professionnelle. Des améliorations en termes de simplicité auraient pu être apportées en consacrant davantage de temps à la réflexion. Dans ce projet, on a jamais utilisé la récursivité dans l'implémentation des fonctions, car la majorité des idées algorithmiques n'étaient pas basées sur ce principe.

---

## 6 Conclusion

Dans le cours "Projet de programmation" de M. Renault en première année d'informatique, nous avons abordé le jeu "Mansuba". L'objectif était de créer une version de base du jeu, dans laquelle deux joueurs s'affrontaient avec le même nombre de pions. Il y avait différentes manières de déplacer les pions et deux conditions de victoire différentes. Le jeu a été développé et amélioré dans le cadre d'autres *achievements*. Au total, chaque développeur a consacré plus de 75 heures au projet et écrit plus de 3000 lignes de code. Pour cela, 18 programmes différents ont été codés et des fonctions ont été vérifiées pour s'assurer qu'elles étaient correctes. Le projet nous a permis d'approfondir notre connaissance en programmation avec le langage C. Nous avons appris à organiser un projet en équipe et à nous concerter. Nous avons également pris conscience de l'importance des tests pour les fonctions écrites.