

Rapport de projet - Semestre 5 - Groupe 17254

FIGURE 2 –

Mansuba

Filière Informatique - ENSEIRB-MATMECA

Auteur :
MEDGHALI Majid

Encadrants :
RENAULT David
DULAU Idris
GUERMOUCHE Amina

Table des matières

1	Introduction	3
2	Outils de travail	3
3	Architecture	3
3.1	Graphe de dépendances	3
3.2	Makefile	4
4	Objets algorithmiques	4
4.1	Struct world	4
4.2	Struct set	5
4.3	Struct neighbors_dir	5
5	Déroulement du projet	5
5.1	Relations & Plateaux	5
5.2	Pièces & Déplacements	7
5.3	Game Over & Boucle principale de jeu	14
6	Tests & Affichages	16
6.1	Tests	16
6.2	Affichages	16
7	Perspectives	17
8	Conclusion	17

1 Introduction

Ce projet a été réalisé dans le cadre d’une évaluation pour le semestre 5. Il s’agit de **Mansuba**, un jeu de dames chinoises visant à mettre en place un ensemble de structures permettant de jouer sur différents plateaux avec des pièces aux mouvements variés. Le projet comprenait plusieurs objectifs qui permettaient de modifier le type de pièces, les plateaux de jeu et les stratégies de jeu.

2 Outils de travail

J’ai travaillé de manière individuelle pendant les séances de projet et j’ai utilisé Git pour gérer mon code. Cet outil me permet de revenir facilement à une version précédente en cas de problème avec la dernière version. Ce rapport est rédigé en *LATEX* qui est un éditeur de texte permet d’écrire facilement des documents scientifiques, et j’ai également utilisé des outils de débogage tels que *GDB* et *Valgrind* pour résoudre des problèmes de segmentation. Les fichiers mentionnés dans ce rapport se trouvent sur mon dépôt *Git*.

3 Architecture

3.1 Graphe de dépendances

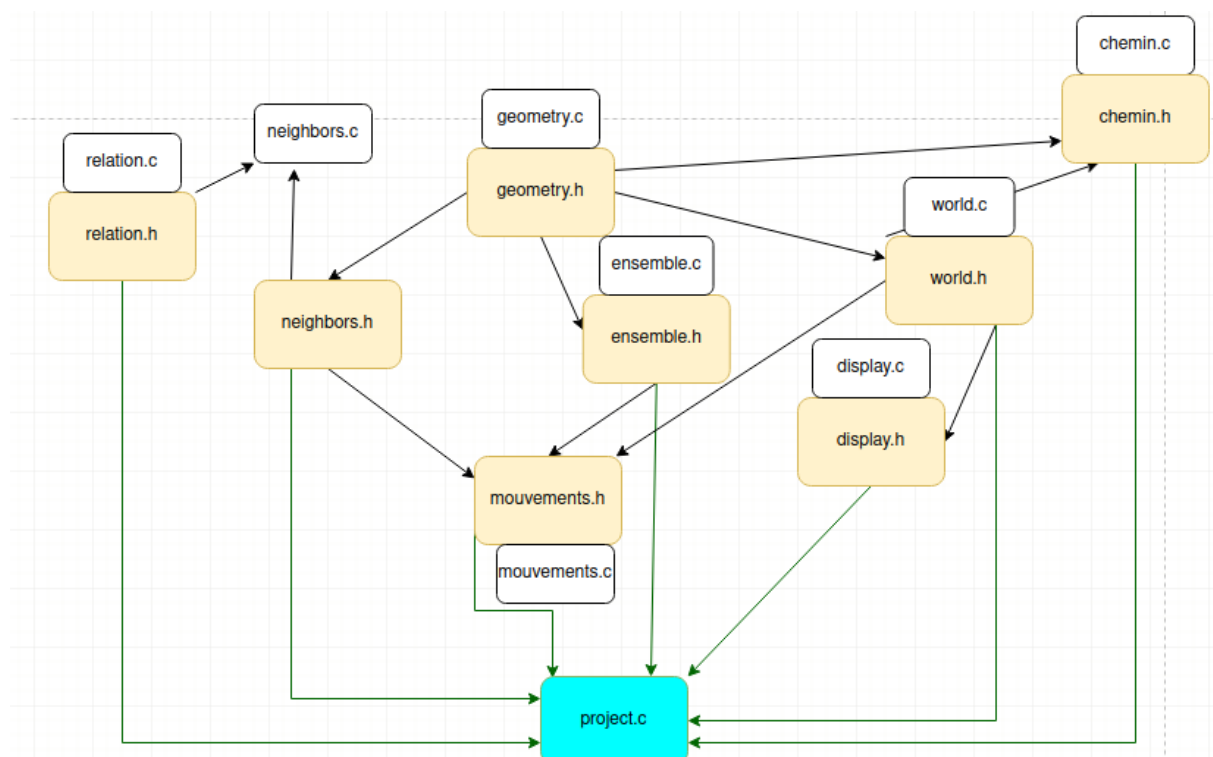


FIGURE 3 – Diagramme de dépendance

Dans cette partie, nous parlerons de dépendances des fichiers, dans la figure 3, les fichiers de base sont `geometry.h` et `neighbors.c` et `world.h`, au début, nous avons une contrainte

sur ces fichiers de ne pas les changer. Dans les *achievements* suivantes, j'ai ajouté les *sorts* *TOUR*, *ELEPHANT*, *HORSE* et *NO_CELL* dans le fichier `geometry.h`, ainsi que les directions `DOUBLE_EAST` et `DOUBLE_WEST`.

Le fichier `ensemble.h` contient la structure *ensemble* et les prototypes des fonctions associées. Le fichier `relation.h` contient une fonction permettant de changer le type du plateau de jeu. Ce fichier contient également une constante nommée *RELATION*, qui correspond au numéro du plateau et peut être modifiée via une ligne de commande (elle est par défaut égale à 2, pour le plateau triangulaire). Le fichier `mouvements.h` contient les prototypes des fonctions liées aux déplacements, tandis que `display.h` contient les fonctions d'affichage. Les implémentations de ces deux fichiers se trouvent dans les fichiers `display.c` et `mouvements.c`.

Le fichier principal `project.c` contient des fonctions permettant d'initialiser le plateau, d'initialiser les joueurs, de vérifier la fin de la partie et la boucle principale de jeu.

Les autres `fichiers.c` chacun contient l'implémentation des fonctions du `fichier.h` associé.

3.2 Makefile

Les fichiers dans le dépôt sont organisés de la manière suivante :

```
/          -- la racine du répertoire du projet
Makefile   -- Makefile global
README.md  -- fichier qui contient les instructions de compilation et d'exécution
/src       -- fichiers sources (.c/h)
/src/test  -- les différents tests
/rapport/  -- Une copie du rapport sous format PDF
```

Le Makefile fournit :

Une règle `all` qui compilera le fichier principal et tous les tests

Une règle `project` qui compilera la boucle principale du jeu et donnera un exécutable `project`.

Une règle `test` qui compilera l'ensemble des tests et donnera deux exécutables `test_set` et `test_1`.

4 Objets algorithmiques

4.1 Struct world

Dans ce sujet, le monde est divisé en `WORLD_SIZE` cases, chaque case étant caractérisée par sa couleur et son *sort*. J'ai décidé de choisir la structure suivante pour décrire le monde.

```
struct place{
    enum color_t c;
    enum sort_t s;
};

struct world_t{
    struct place t[WORLD_SIZE];
};
```

4.2 Struct set

Pour stocker les positions des pièces de jeu et leurs mouvements, le sujet nous demande de choisir une structure. Plutôt que d'utiliser un tableau de `WORLD_SIZE` éléments et d'écrire une fonction pour déterminer le nombre de mouvements possibles ou de pièces de jeu, j'ai opté pour une structure comprenant un tableau et un indice correspondant au nombre de cases remplies. Cela nous permet également de simplifier la complexité.

```
struct set{
    int t[WIDTH*HEIGHT];
    int size;
};
```

J'ai implémenté plusieurs fonctions associées à la structure `set`, notamment l'ajout et la suppression d'un ou plusieurs éléments, ainsi que la vérification de l'état de vide de la structure et la création d'un ensemble vide. Toutes ces informations se trouvent dans les fichiers `ensemble.c` et `ensemble.h`.

4.3 Struct neighbors_dir

```
struct neighbors_dir{
    enum dir_t d[MAX_NEIGHBORS+1];
};
```

Le fichier `neighbors.c` contient la structure `neighbors_dir` conçue pour mémoriser les directions des voisins de chaque case. Pour ce faire, j'ai créé un tableau `struct neighbors_dir tab[WIDTH*HEIGHT]` au début de même fichier et la fonction `init_neighbors` s'occupera de remplir ce tableau avec les directions des voisins de chaque case.

5 Déroulement du projet

5.1 Relations & Plateaux

Le changement d'un plateau revient au changement des relations entre les cases, comme pour certaines relations, nous pouvons avoir des cases qui n'ont pas de voisins donc je change les *sorts* de ces cases par `NO_CELL`.

l'initialisation du plateau revient à mettre `NO_CELL` comme *sort* pour les cases qui n'ont pas de voisins, on fait cela avec la fonction `initialize_plataeu`.

Plateau carré

Au début, nous avons travaillé avec un plateau carré, donc pour chaque case, les voisins possibles se trouvent dans les huit directions de la figure5. Dans la fonction `init_neighbors` je stocke pour chaque case les directions de ses voisins dans la structure `neighbors_dir`, ensuite dans la fonction `get_neighbors`, nous parcourons ces directions et obtenons les voisins possibles grâce à la fonction `get_neighbor`. Pour tester les fonctionnalités de ces deux fonctions, j'ai inclus les fonctions de test `test_get_neighbor` et `test_get_neighbors` dans le fichier `tests_1.c` afin de minimiser les risques de problèmes dans la boucle principale, la figure 4 montre la forme du plateau carré.

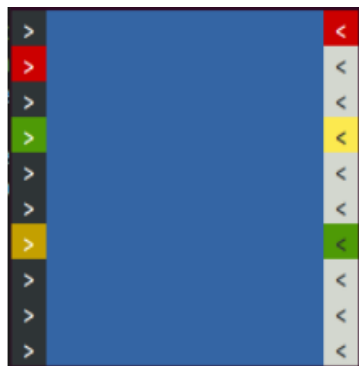


FIGURE 4 – la forme du plateau carré

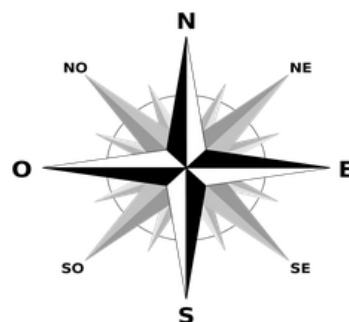


FIGURE 5 – Les directions des voisins du plateau carré

Plateau triangulaire

Pour réaliser le plateau triangulaire, j'ai dû ajouter des directions supplémentaires, comme *DOUBLE_EAST* et *DOUBLE_WEST*, qui représentent respectivement les cases situées deux cases à gauche et deux cases à droite la case actuelle. La figure 6 montre les directions de la relation et la forme du plateau.

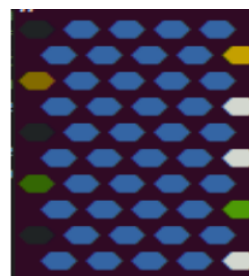
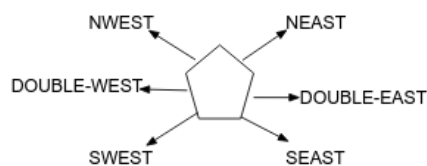


FIGURE 6 – Les directions des voisins et la forme du plateau triangulaire

Plateau de 4 voisins

Sur ce plateau, chaque case a au maximum quatre voisins dans les directions de la figure 7.

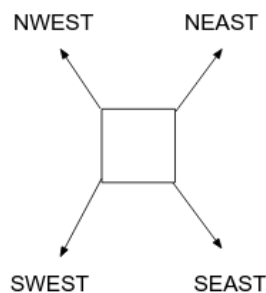


FIGURE 7 – Les directions des voisins

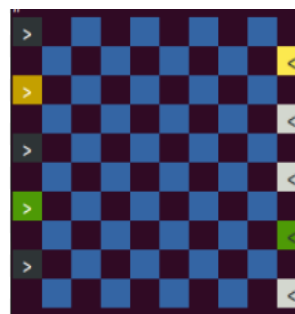


FIGURE 8 – La forme du plateau de 4 voisins

5.2 Pièces & Déplacements

Dans cette partie, nous parlerons des différentes pièces du projet et leurs déplacements sur les différents plateaux et aussi sur les problèmes rencontrés.

Un pion simple :

Les déplacements des pions sont regroupés en 3 types :

Déplacement simple : Le pion se déplace vers un voisin inoccupé selon la relation définie.

Pour réaliser cela, j'ai codé la fonction `possible_neighbors` qui utilise `get_neighbors` et retourne les voisins libres.

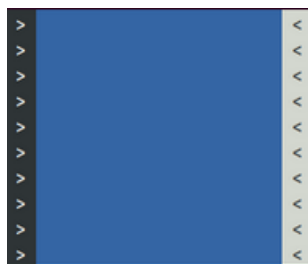


FIGURE 9 – Le plateau à l'instant t



FIGURE 10 – Le déplacement du pion blanc à l'instant $t+1$

Saut simple : Le pion saute par-dessus un pion de couleur différente vers une case libre.

Pour déterminer les sauts possibles, j'ai implémenté la fonction `possible_jumps` qui utilise les voisins occupés et teste la possibilité de sauter par-dessus ces derniers.



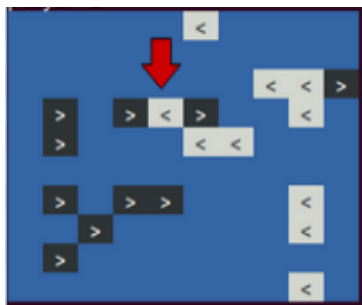
FIGURE 11 – le pion noir à l'instant t



FIGURE 12 – le pion noir à l'instant $t+1$

Saut multiple : Le pion réalise plusieurs sauts consécutifs pour atteindre une case libre.

Pour déterminer les multiples sauts, j'ai créé une boucle dans laquelle j'applique à chaque itération la fonction `possible_jumps` aux sauts précédemment trouvés afin de trouver de nouveaux sauts possibles. J'ai choisi 4 comme nombre maximal de sauts que nous pouvons effectuer, et à la fin la fonction `possible_moves` retourne l'ensemble des mouvements possibles, c'est-à-dire les sauts possibles et les voisins libres. Les deux figures figure 13 et figure 14 montrent un multiple saut du pion blanc.

FIGURE 13 – Le pion blanc à l'instant t FIGURE 14 – Le pion blanc à l'instant $t+1$

Problème rencontré : La façon dont j'ai codé les multiples sauts consiste à chercher à chaque étape les sauts possibles à l'état $t+1$ à partir des sauts trouvés à l'état t . Ainsi, l'exemple de la figure 15 illustre le problème que j'ai rencontré.

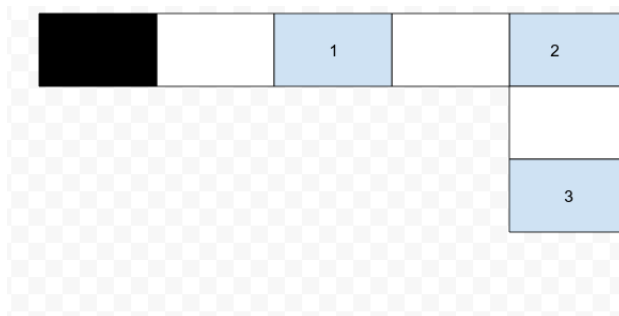


FIGURE 15 – problème des multiples sauts

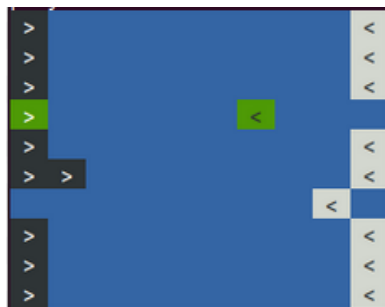
Selon la figure 15, les cases $\{1,2\}$ sont accessibles par des sauts pour le joueur noir. Cependant, en utilisant l'algorithme de sauts multiples, nous trouverons également la case $\{3\}$ comme destination possible, car les sauts possibles à l'étape 2 sont $\{1,2\}$, ce qui nous permet de sauter sur la case $\{3\}$ depuis la case 2 lors de l'étape 3. Pour éviter ce problème, nous devons ajouter une condition qui interdit de chercher des sauts depuis une case située dans les positions initiales de l'adversaire.

Pour généraliser les déplacements des pièces, nous avons ajouté d'autres pièces de mouvements différents. parmi ces pièces, nous trouvons :

La tour

La tour peut se déplacer sur un nombre illimité de cases dans les directions cardinales, à condition qu'elle ne soit pas bloquée par une autre pièce sur son parcours.

Pour déplacer une tour, nous utilisons la fonction `tour_moves` qui nous indique les mouvements possibles pour cette tour, les deux figures figure16 et figure17 montrent un déplacement de la tour verte sur le plateau carré.

FIGURE 16 – le plateau à l'instant t FIGURE 17 – Déplacement de la tour verte à l'instant $t+1$

L'ÉLÉPHANT

L'éléphant peut se déplacer sur toutes les positions libres à partir de sa position en suivant deux directions cardinales.

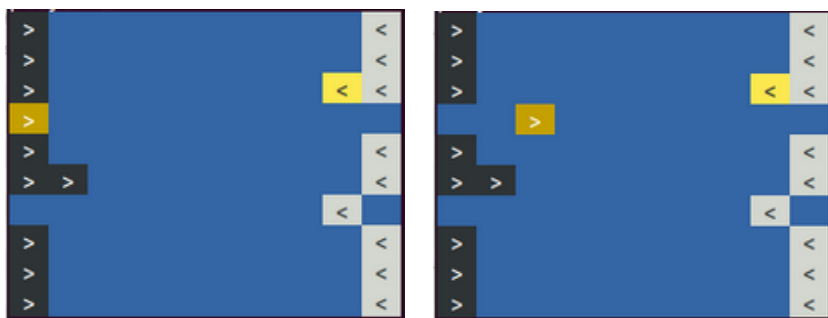


FIGURE 18 – Déplacement de l'éléphant jaune

Pour déplacer un éléphant, nous utilisons la fonction `elephant_moves` pour connaître les mouvements possibles.

Le CAVALIER

Le cavalier joue sous forme de la lettre **L**, Les mouvements possibles pour un cavalier sont donnés par la fonction `horse_moves` la figure figure 19 montre un déplacement du cavalier rouge.

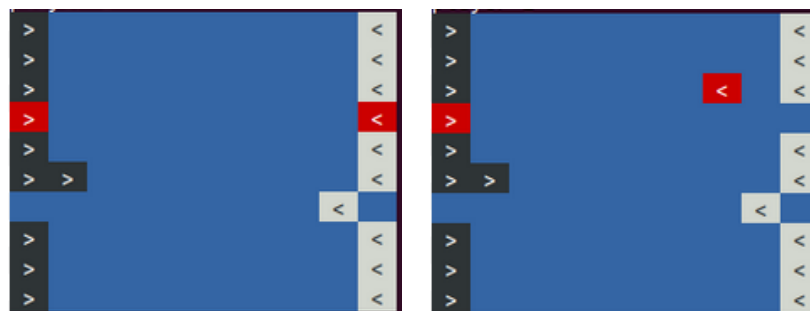


FIGURE 19 – Déplacement du cavalier rouge

Remarque

J'ai rencontré un problème avec les pièces de type cavalier et de type tour (en rouge et en vert) qui ne peuvent être utilisées que sur un plateau carré. Cela est dû au fait que le cavalier se déplace en forme de lettre L, ce qui signifie que tous ses mouvements tombent sur des cases qui n'ont pas de voisins. De même, la tour peut jouer, mais elle n'atteint jamais les positions initiales de l'adversaire. Ainsi, si le type de plateau choisi est différent du plateau carré, il n'y a aucun cavalier ni aucune tour sur la partie.

Initialisation des joueurs & Stratégies de jeu

Initialisation des joueurs

Pour pouvoir jouer une partie, nous avons besoin d'au moins deux joueurs. Pour identifier ces joueurs, j'ai créé la structure suivante :

```
struct player{
    enum color_t color;
    int dir;
    struct set current_places;
    struct set begining_places;
};
```

Chaque joueur est caractérisé par sa couleur (noire ou blanche), sa direction (1 ou 2) et deux structures. La première structure est utilisée pour enregistrer les positions actuelles des pièces du joueur et utilisée lors des déplacements de ces dernières. La seconde structure enregistre les positions initiales des pièces du joueur et est utilisée pour tester la fin de la partie.

Dans la fonction *initialize_player*, nous remplissons les structures *begining_places* et *current_places* avec les cases de la première ou de la dernière colonne en fonction de la direction du joueur. Nous remplissons également les couleurs en fonction de la direction du joueur et effectuons les changements nécessaires des *sorts* et des couleurs sur le monde. chaque joueur commence la partie avec une *TOUR*, un *CAVALIER*, un *ELEPHANT* et les autres pièces sont tous des *PAWNS*.

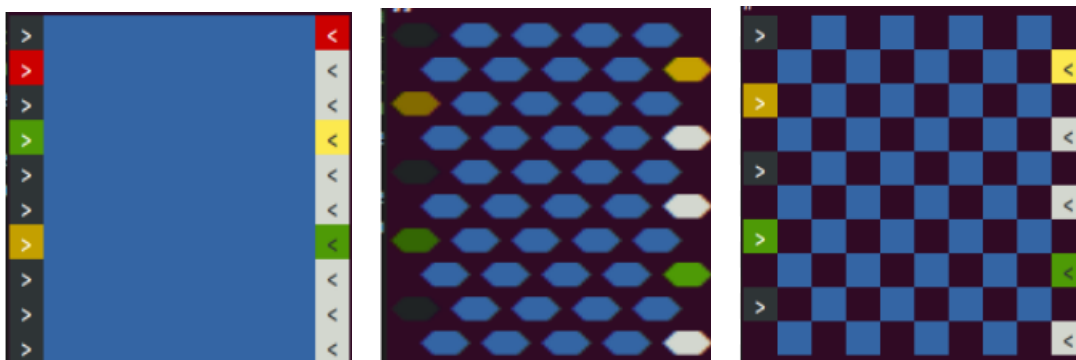


FIGURE 20 – L'initialisation des joueurs sur les 3 plateaux

Les pièces en rouge sont des cavaliers, en vert sont des tours et en jaune sont des éléphants.

Capture des pièces

Au départ, nous avons commencé par un jeu aléatoire dans lequel chaque joueur sélectionne une pièce de manière aléatoire parmi celles disponibles, et choisit sa destination aléatoirement parmi les destinations possibles. Le but est d'arriver aux positions initiales de l'adversaire, tout cela est réalisé à travers la fonction `play`. Jusqu'à présent, le nombre de pièces de chaque joueur est resté statique. Pour donner la possibilité de changer ce nombre, nous avons permis aux pièces de capturer d'autres pièces. Chacune des pièces capturées a également la possibilité de s'évader avec une probabilité de 50% si la position où elle a été capturée est libre. Le premier problème rencontré était de savoir comment mémoriser les positions et les *sorts* des pièces capturées. Pour résoudre ce problème, j'ai ajouté la structure suivante :

```
struct cell_preson{
    int type;
    struct sort sort;
    struct sort first_sort_died;
};
```

et j'ai aussi ajouté dans la structure *player* le tableau suivant :

```
struct cell_preson t[HEIGHT*WIDTH];
```

Avec ce tableau, le joueur peut savoir l'état de ses pièces. Chaque pièce du joueur est associée à une structure `cell_preson` qui nous permet de savoir si elle est prisonnière ou non en fonction de l'entier `type`. On peut aussi mémoriser son *sort* si elle est capturée grâce au champ `sort`. J'ai limité le nombre de pièces d'un même joueur qui peuvent être capturées à la même position à 2. Ainsi, le *sort* de la pièce qui a été prisonnière en premier est `first_sort_died`.

Pendant la partie, si la pièce choisie est prisonnière et que sa position est libre, la pièce peut être s'évader avec une probabilité de 50%, alors pour réaliser cela, je modifie l'entier `type` dans la structure `cell_preson` associée à la pièce en question et je mets à jour dans le monde la couleur du joueur et le champ `sort` de la structure. La suivante figure 21 illustre un exemple de pièce qui a réussi à s'évader.



FIGURE 21 – Une case qui a réussi à s'évader

Dans le cas où la pièce choisie n'est pas prisonnière et que son mouvement tombe sur une pièce appartenant à l'adversaire, nous pouvons alors capturer cette dernière. J'ai mis en place cette fonctionnalité en modifiant l'entier `type` de la structure `cell_preson` correspondante pour le joueur adverse. La figure 22 illustre un exemple de capture de l'adversaire.



FIGURE 22 – Capturer l'adversaire

Plus court chemin

L'objectif de chaque partie, c'est d'atteindre les positions initiales de l'adversaire, donc le fait que les pièces soient déplacées de manière aléatoire ne rend pas les parties très intéressantes, donc j'ai développé une stratégie dans laquelle le joueur joue la case qui le rapproche le plus

possible des cases initiales libres de l'adversaire. Pour ce faire, il est nécessaire de calculer les distances. Pour cela, j'ai créé le tableau suivant :

```
struct set T[WIDTH*HEIGHT];
```

avec struct set

Ainsi, ce tableau est rempli au début de la partie grâce à la fonction `tab_distance`. Il a la forme d'une matrice de longueur `WIDTH*HEIGHT` et de largeur `WIDTH*HEIGHT`, où à l'indice (i,j) se trouve la distance entre i et j calculée en utilisant les relations données. Je remplis le tableau au début de la partie afin d'éviter de devoir le recalculer à chaque fois. La distance est calculée de la manière suivante :

Pour déterminer la distance entre les points i et j , je commence par vérifier si le sort de i ou de j égale à `NO_CELL` donc la distance égale à -1, sinon si j se trouve parmi les voisins de i , alors la distance est égale à 1. Sinon, je recalcule les voisins de l'ensemble formé par i et ses voisins, et je vérifie de nouveau si j s'y trouve. Si c'est le cas, alors la distance est de 2. Si j n'est toujours pas trouvé, je répète cette opération jusqu'à ce que j'aie vérifié tous les points atteignables depuis i .

Ainsi, la case la plus proche est calculée par la fonction `best_move`, qui utilise la structure des mouvements, la structure des positions initiales de l'adversaire et la structure du monde. Cette fonction cherche le meilleur mouvement vers les positions initiales libres de l'adversaire en utilisant la fonction `minimal_distance`, qui prend une case et un ensemble en entrée et renvoie la distance la plus courte entre cette case et l'ensemble, la figure suivante figure 23 montre un exemple de bon saut pour gagner la partie, la figure 24 illustre le trajet le plus proche de la pièce noire pour arriver à la case vide.



FIGURE 23 – Un exemple du chemin minimal

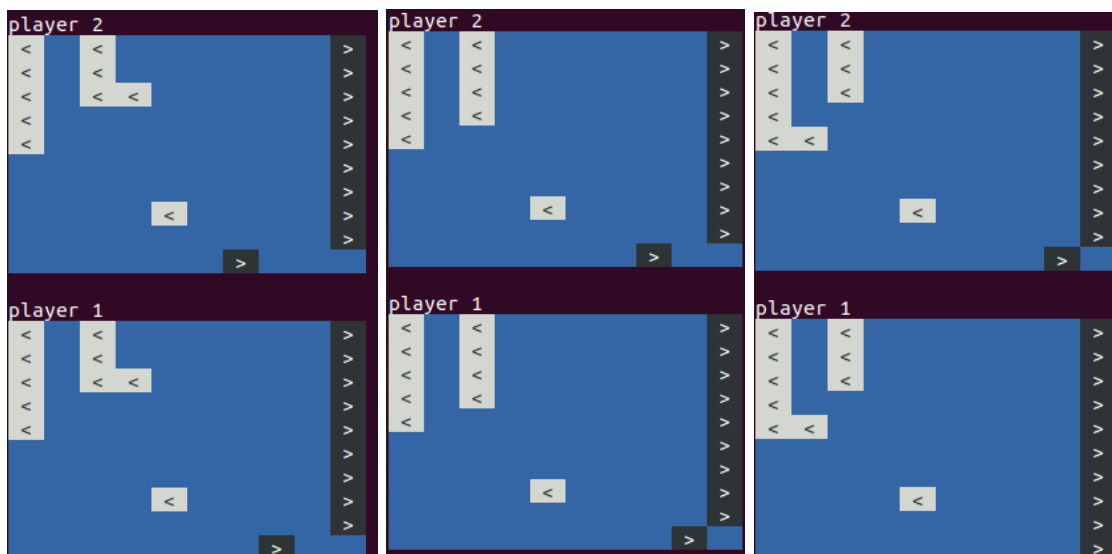


FIGURE 24 – La case noire suit son plus court chemin

5.3 Game Over & Boucle principale de jeu

Dans cette partie, nous parlerons des deux types de victoires et de la construction de la boucle principale.

Game Over

La partie peut se jouer avec deux types de victoire :

Victoire simple : si l'un des joueurs parvient à amener une de ses pièces sur une position initiale de l'adversaire avant MAX_TURNS tours, la figure 25 montre un exemple d'une victoire simple du joueur noir.



FIGURE 25 – Victoire simple

Victoire complexe : Le joueur qui parvient à occuper toutes les positions de départ de son adversaire avec ses pièces avant MAX_TURNS tours remporte la partie, la figure 26 montre

une victoire complexe du joueur noir.

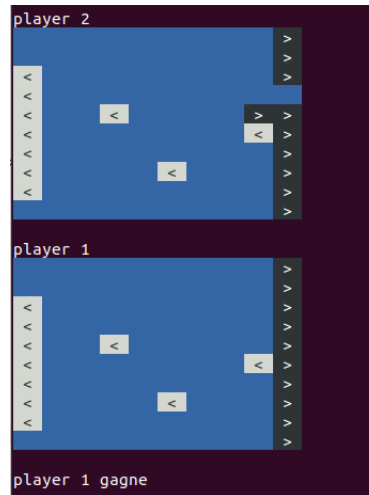


FIGURE 26 – Victoire complexe

Pour mettre en œuvre ces deux victoires, j'ai créé la fonction `win_game` qui teste la fin de la partie en utilisant les positions actuelles du joueur qui joue et les positions initiales de l'adversaire et un indice correspond au type de la victoire et retourne le résultat.

Remarque

il est possible que la partie soit nulle, indépendamment du nombre d'itérations. Voici un exemple de ce type de situation :



FIGURE 27 – Une partie toujours nulle

Les pièces rouges sont des cavaliers et se déplacent en forme de lettre *L*, ce qui signifie qu'ils ne peuvent jamais atteindre les positions libres.

La boucle principale de jeu

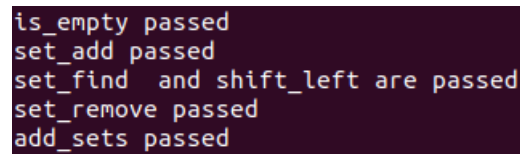
Pour commencer la boucle de jeu, nous devons tout d'abord initialiser le monde avec la fonction `world_init`, remplir la structure `struct set T[WIDTH*HEIGHT]` avec toutes les distances possibles en utilisant la fonction `tab_distance`, initialiser les joueurs avec la fonction `initialize_player` et choisir aléatoirement qui commence en premier. Après cela, chaque joueur joue son tour avec la fonction `play` qui sélectionne aléatoirement une pièce parmi `current_places` et choisit sa destination vers la case le plus proche des positions initiales libres de l'adversaire parmi l'ensemble retourné par `possible_moves`. À la fin de chaque tour,

nous testons avec la fonction `win_game` si l'un des deux joueurs a gagné, nous sortons de la boucle, sinon nous continuons jusqu'à ce que `MAX_TURNS` soit atteint..

6 Tests & Affichages

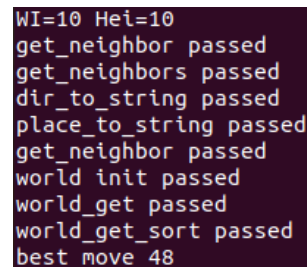
6.1 Tests

Pour s'assurer que nos fonctions fonctionnent correctement, j'ai mis en place deux fichiers de test en utilisant la bibliothèque `assert`. Le premier fichier est `test_set.c` la figure 28, qui permet de tester les fonctions du fichier `ensemble.c` qui ont un lien avec la structure `set`, et `test.c` la figure 29, qui sert à tester les autres fonctions du projet.



```
is_empty passed
set_add passed
set_find and shift_left are passed
set_remove passed
add_sets passed
```

FIGURE 28 – Les tests associés à struct set



```
WI=10 Hei=10
get_neighbor passed
get_neighbors passed
dir_to_string passed
place_to_string passed
get_neighbor passed
world init passed
world_get passed
world_get_sort passed
best move 48
```

FIGURE 29 – Les autres fonctions testées

6.2 Affichages

Pour pouvoir visualiser l'avancement de la partie, j'ai réalisé deux affichages, le premier est un affichage sur le terminal et l'autre est un affichage `sdl`, les deux affichages sont dans la figure 30.

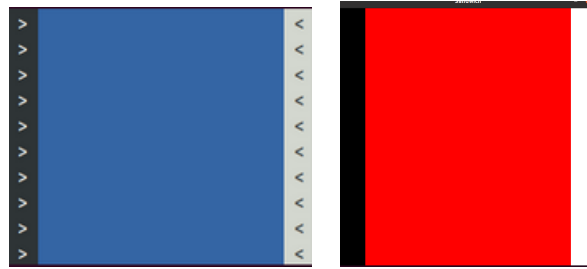


FIGURE 30 – L’affichage terminal et l’affichage sdl

J’ai rencontré un problème pour afficher la forme d’un hexagone sur SDL, donc pour le plateau triangulaire, j’ai travaillé seulement avec l’affichage terminal.

7 Perspectives

Nous pouvons faire plusieurs améliorations possibles dans ce projet :

Possibilité d’améliorer les multiples sauts au lieu de faire 4 sauts au maximum, faire tous les sauts possibles.

Possibilité d’améliorer l’affichage `sdl` pour afficher la forme hexagone.

Améliorer le calcul de la distance en fonction du nombre de coups nécessaires à la pièce pour passer de la première position à la seconde, plutôt que de calculer la distance entre deux positions en comptant le nombre de déplacements simples entre ces deux points.

8 Conclusion

Ce projet m’a beaucoup appris, notamment en ce qui concerne l’utilisation de l’environnement `Unix` et l’écriture en *LATEX*. J’ai notamment appris à écrire un `Makefile` de C, comment utiliser efficacement les pointeurs et les structures, et comment faire les bons choix de structures pour réaliser ce qui est demandé. La liberté offerte par le sujet et les nombreuses possibilités qu’il offre nous ont permis de créer des simulations amusantes et intéressantes, comme le choix des plateaux de jeux, ainsi que de choisir les pièces. La découverte de la responsabilité de travailler individuellement et de chercher soi-même les solutions aux différents problèmes, s’est avérée être une expérience enrichissante. Le bilan est positif, tant en ce qui concerne les choses apprises que les travaux accomplis.