Rapport de projet S5:

MANSUBA

Réalisé par:
MOHAMMED BOUHAJA ET AMIRA ELOUAZZANI
Encadré par: Julien Allali

January 11, 2023

Contents

1 INTRODUCTION

1.1 Problèmatique

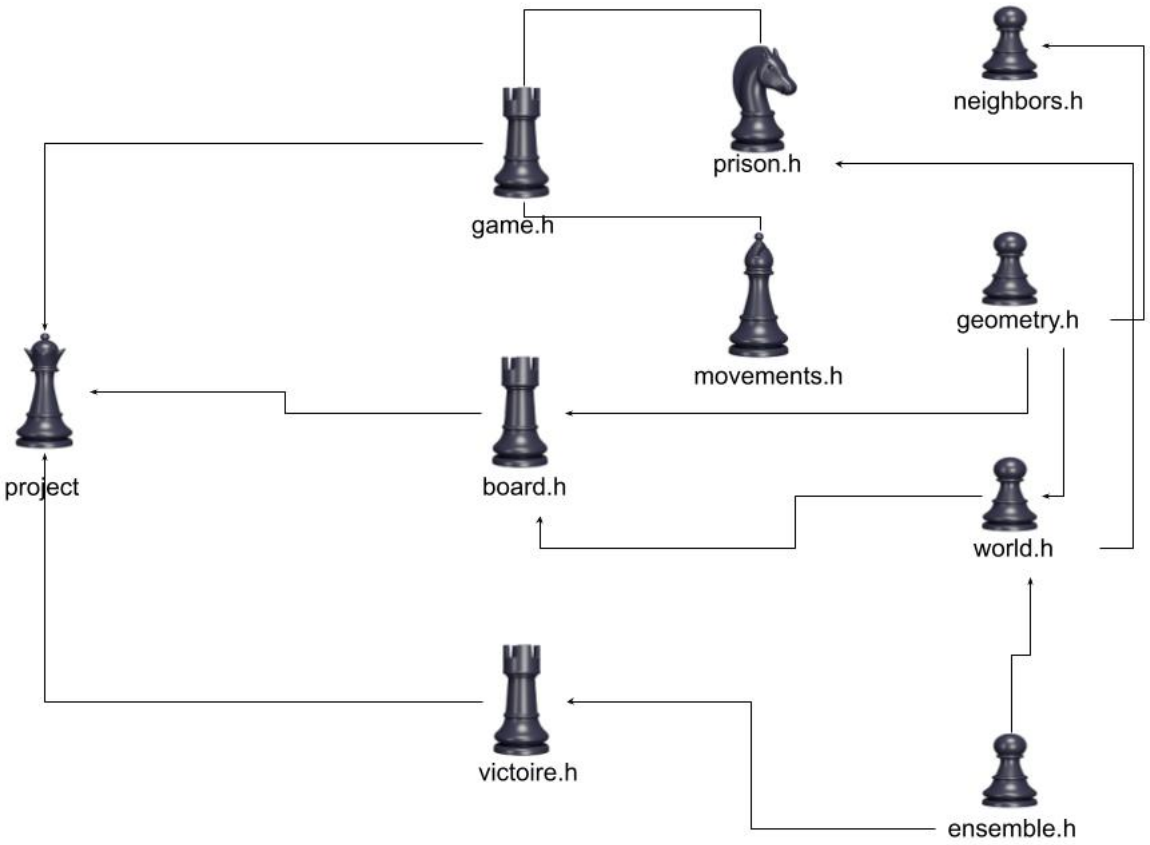
Mansuba est un jeu de plateau , ancêtre de Shtranj , qui a comme but mettre l'autre joueur en situation de mat . Le but de notre projet sera de jouer une partie aléatoire, puis rendre l'algorithme de plus en plus flexible et l'orienter finalement vers la victory.

1.2 Hypothèse et démarches de validation

Pour jouer une partie, il faut que le plateau de jeu soit définie au préalable. Le plateau de jeu *board* étant le binôme monde et relation. Le monde *world* est l'set des cases accessibles par les pions et dont les mouvements seront permis . Chaque case a désormais des cases voisines *Neighbors* dans des directions précisé par `enum dir_t`.

```
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
};
```

La possibilité d'accès à ses voisins sera contrôlée par la relation de la partie et qui choisira parmis les voisins ceux qui sont accessibles par mouvement direct.



2 Bases du jeu

Le jeu s'effectue un monde de WORLD_SIZE = WIDTH × HEIGHT case , des pièces (`enum sort_t`) pour les joueurs ayant la couleur dans (`enum color_t`) . Cette configuration est surtout gouvernée par *geometry.h*.

```
enum sort_t {
  NO_SORT  = 0,   // Default sort (i.e nothing)
  PAWN     = 1,
  TOUR     = 2,
  ELEPHANT = 3,
  MAX_SORT = 4,   // Total number of different sorts
};
enum color_t {
  NO_COLOR  = 0,  // Default color, used to initialize worlds
  BLACK     = 1,
  WHITE     = 2,
  MAX_COLOR = 3,  // Total number of different colors
};
```

Pendant le développement du jeu on fera souvent appel à toute la configuration du jeu. On rassemble alors tous les paramètres du monde actuel pendant le jeu dans une structure `struct`

game_t.

```
struct game_t {
    enum color_t current_player;
    unsigned int tour;
    struct world_t* w;
    struct jail_t* jail;
    unsigned int seed;
    unsigned int position;
    enum victory_t victory;
};
```

Pour jouer une partie aléatoire on aura besoin de configurer le plateau de jeu.

2.1 Partie monde

La structure world est un tableau de pair couleur et type de pions qui inaccessible que par des fonctions de *world.h*. On commence d'abord par initier un monde sans aucun pion à l'aide de la structure `struct world_t* world_init();`. Cette fonction attribue à chaque case du tableau monde le pair (NO_COLOR , NO_SORT). Les fonctions de world.h permettront l'écriture et la lecture de la couleur et du type d'une case donné et seront utilisé fréquemment pour accéder au monde.

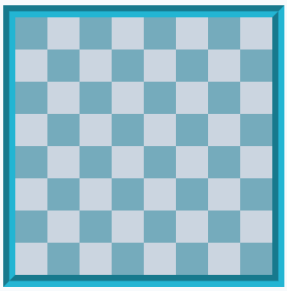


Figure x: Un monde avec 64 cases indexées de 0 à 63.

2.2 Partie relation

La structure `struct neighbors_t` prédéfinira les voisins de chaque indice donné .

```
struct neighbors_t {
  struct vector_t n[MAX_NEIGHBORS+1];
};
```

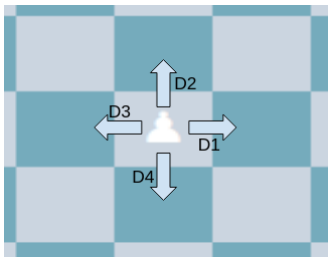


Figure x:

Les voisins seront un tableau dont le contenu pour un indice donné est un tableau de vecteur introduit par la structure `struct vector_t` défini par l'indice du voisin et sa direction de taille MAX_NEIGHBORS . De la même façon que world, les fonctions qui donne accès au voisins sont

`struct neighbors_t get_neighbors(unsigned int idx)` qui cherche dans la structure des voisins l'élement d'indice rentré en paramètre et `unsigned int get_neighbor(unsigned int idx, enum dir_t d)`qui aide à obtenir le voisin dans une direction donné par des opérations sur l'indice.

L'initialisation d'une relation modifie la liste des voisins pour qu'il puisse inclure que des voisins de certain directions donné. Avant d'initialiser une relation on pose dans la structure neighbors comme premier voisin pour chaque indice le pair (UINT_MAX , NO_DIR) et un (0,0) pour le reste. UINT_MAX est définie dans *limits.h*. La fonction add_neighbor servira par suite à pousser (UINT_MAX , NO_DIR) et le remplacer par le pair indice du voisin et sa direction.

2.3 set

```
struct set{
    unsigned int taille;
    unsigned int positions[WORLD_SIZE];
};
```

Cette structure `struct set` permettra de construire des tableaux d'une taille donnée et simplifier leurs manipulations : lecture et écriture , comme le jeu a plusieurs collections à fournir : l'set des positions des pions, la collection des mouvement possibles (qui sera le but de l'étape qui suit) ... . Elle contient comme attribut un entier qui donne la taille et un tableau d'indice qui sont le contenu de l'set. On a défini en plus une fonction qui sera utile pour le reste, add_position, qui augmente la taille et remplace la position d'indice rentré en paramètre par sa valeur.

```
void add_position(struct set* p ,unsigned int place ){
    p->positions[p->taille]=place ;
    p->taille+=1;
}
```

3 Mobilités des pièces

3.1 Mobilité d'un pion *PAWN*

Pour stocker les mouvements on fera appel à la structure set . Les mouvement possibles dans la version de bases sont :

Déplacement simple : pour les relever il suffit d'utiliser la fonction get_neighbors pour un indice donné . Ils seront stockés par la fonction `void simple_moves( struct game_t game, struct set* sm );`

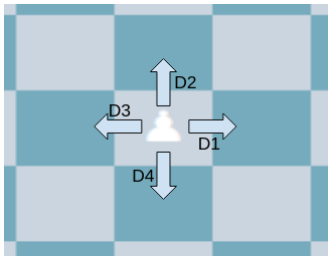


Figure x: Déplacement simple d'un pion.

Saut simple : si le voisin dans une direction j est occupé, équivalent d'avoir la fonction get_neighbor dans la direction j qui a un SORT différent de NO_SORT, et le voisin du voisin dans la meme direction est libre , on peut se déplacer vers le voisin du voisin , si le monde le permet. Ils seront stockés par la fonction `void simple_jumps(struct game_t game , struct set* sj);`

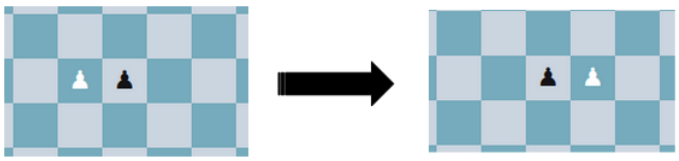


Figure x: Saut simple d'un pion.

Saut multiple : s'agit d'une suite de saut simple . Cette fonction avait besoin d'une condition d'arrêt pour éviter de boucler infiniment sur les sauts simples autre que les deux voisins soient occupés. Pour cela il nous fallait une fonction qui vérifie l'existence d'un élément dans un set donné `int place_visited(struct set* ens, unsigned int place );`Elle renvoie un 0 si la place n'est pas encore mentionnée dans l'set qui sera notre condition pour arrêter la recherche. Ils seront stockés par la fonction `void multiple_jumps(struct game_t game , struct set* mj );`.

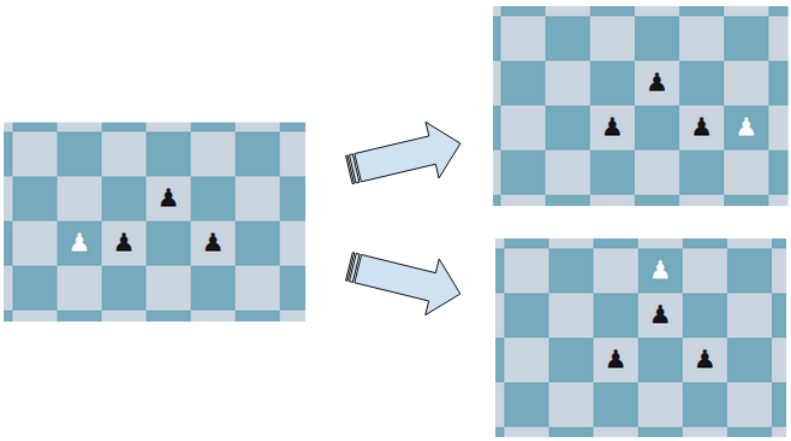


Figure x: Saut multiple d'un pion.

Ses fonctions prennent comme paramètre additionnel un set. Il servira comme espace de stockage pour chacune des fonctions au lieu de retourner l'set à la fin de chacune . En plus, la fonction `void available_movements(struct game_t game, struct set* am)`fera appel à toutes les fonctions de mouvement citée au-dessus et l'set rentrer comme paramètre dans cette fonction sera le même rentrer dans tous les fonctions pour qu'on puisse stocker tous les mouvements dans un même set.

3.2 Mobilité des autre pièces *TOWER* et *ELEPHANT*

3.3 Possibilité de capture

4 Boucle de jeu

Le monde étant inacessible par d'autre document autre que *world.h*. On a utilisé les fonctions `void world_set(struct world_t* b, unsigned int idx, enum color_t c)` et `void world_set_sort(struct world_t * b, unsigned int idx, enum sort_t c)` pour donner à un monde initialisé vide des positions pour chacun des pions. Le nombre des pions étant HEIGHT.

Il existe deux types de victorys. La victory est dite simple si le joueur atteint une des positions de départ de l'autre joueur et complexe si le joueur les atteint tous. En tous cas, on aura besoin de garder la liste de positions de départ des deux joueurs et leurs faire rentrer en paramètre pour pouvoir comparer avec les positions actuelles de l'adversaire. En plus, pour la comparaison on pourra utiliser la fonction place_visited.

Le jeu s'arrête si on atteint une victory, le choix du type étant aléatoire pendant la partie, ou si on atteint MAX_TURNS.

Avant d'obtenir la boucle de jeu finale il faudra définir des fonctions qui font des choix aléatoires sur tous les paramètres du jeu. Le choix du pion sera fait par `void choose_random_piece_belonging_to( struct game_t* game)`qui retournera un indice. La fonction `unsigned int choose_random_move_for_piece (struct game_t game)` va chercher entre les mouvements disponibles pour cet indice et va ensuite choisir un indice auquel le pion va bouger. La fonction `void move_piece(struct game_t game, unsigned int dst)` agira sur le monde et échangera l'état de la case à l'indice initial avec celle de l'indice du mouvement. On choisira ainsi aléatoirement le premier joueur à commencer.

Dans cette partie on cherche à rendre le jeu de plus en plus générique pour qu'il nous offre plus de possibilités. La version initaile (2) est un peu basique car il contient le même type de pièces avec les mêmes déplacements. Alors dans la suite on va essayer en premier temps de définir plusieurs types de pièces(Tour, Éléphant) avec de nouveau mouvements possibles(translation cardinal et saut semi-diagonal), avant de passer , et finalement ajouter la possibilité de capturer les pièces du joueur adversaire pour que la partie soit plus intéréssante et amusante.