Rapport de projet S5:

MANSUBA

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

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 victoire.

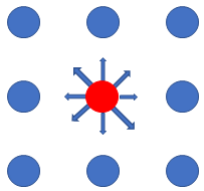
1.2 Hypothèse et démarches de validation

Pour jouer une partie, il faut que le plateau de jeu soit définie au préalable. Le plateau de jeu *board* est le binôme monde et relation. Le monde *world* est l'ensemble 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.

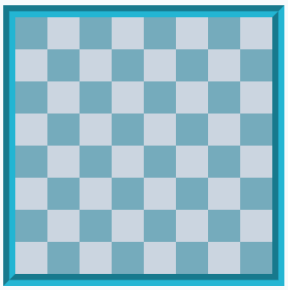


Figure x: Un monde avec 64 cases indexées de 0 à 63.

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];
};
```

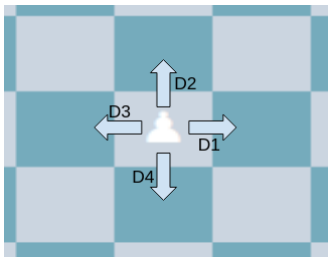


Figure x:

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. Le paramètre *seed* permettera de désigner la relation de la partie et sera modifiable à $\sqrt{\text{MAX\_TURNS}}$. On donne le choix entre 4 relations : simple, diagonale, triangulaire, hexagonal.

Le passage d'une relation à une relation hexagonal est un peu difficile à gérer puisque l'emplacement des pions n'est pas similaire .

2.3 Ensemble

```
\subsection{Ensemble}
\begin{lstlisting}

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 );`

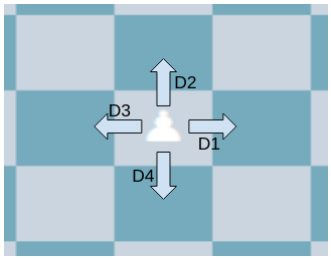


Figure x: Déplacement simple d'un pion.

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);`

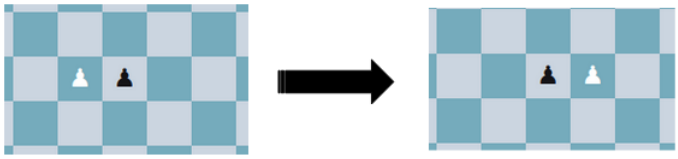


Figure x: Saut simple d'un pion.

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 );`.

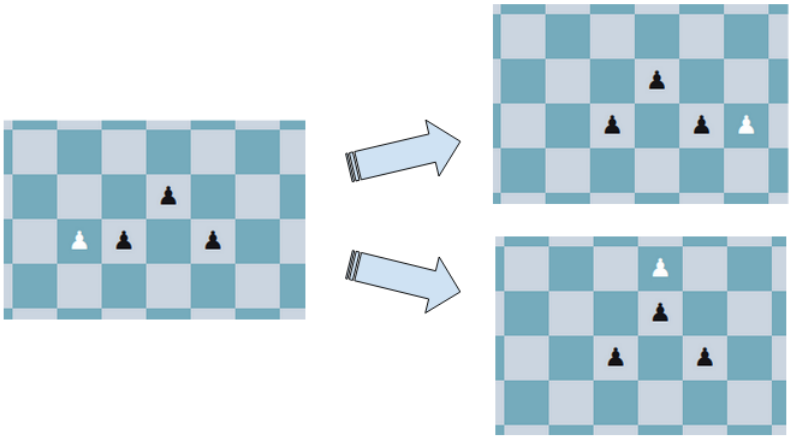


Figure x: Saut multiple d'un pion.

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*

Translation cardinale : La tour se déplace en ligne droite soit horizontalement soit verticalement de tout nombre de cases inoccupées, donc on réalise une boucle sur les quatres direction SOUTH, NORTH, EAST et WEST, et on ajoute les position libre dans l'ensemble *ct* de la fonction `void cardinal_translations(struct game_t game, struct set* ct)` jusqu'à ce qu'elle atteigne le bord de l'échiquier ou qu'elle soit bloquée par une autre pièce. Elle ne peut passer au dessus d'une autre pièce.

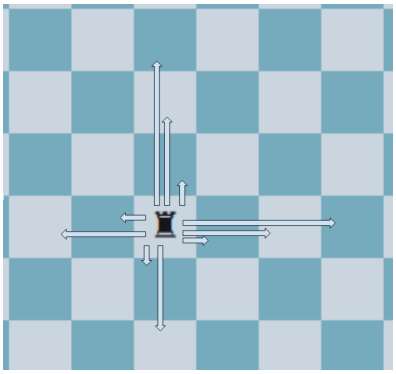


Figure x: Translation cardinal d'une tour.

Saut semi diagonal : L'éléphant peut se déplacer sur les deux diagonale à partir de sa position initial, pour ce faire on boucle sur les directions $i+j$ ($o\grave{u}$ $i \in \{NORTH,\ SOUTH\}$ et $j \in \{EAST,\ WEST\}$) et on ajoute les positions libres. finalement on stocke le résultat dans un ensemble *sdj* à l'aide de la fonction `void semi_diagonal_jumps(struct game_t game , struct set* sdj)`.

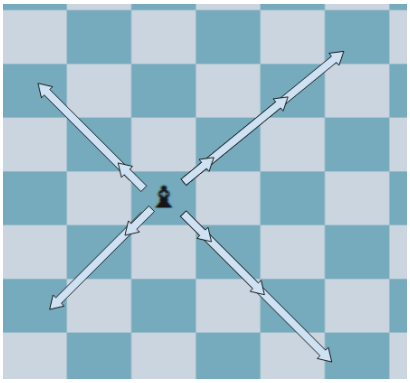


Figure x: Saut semi diagonal de l'éléphant.

3.3 Possibilité de capture et les tentatives d'évasion

Les mobilités des pièces ont un rôle très important dans notre jeu, mais ceci rend le jeu un peu statique où le nombre de pièces reste le même du début jusqu'à la fin. Donc on va améliorer ça en ajoutant d'une part des captures simples qui permettent par exemple dans un jeu d'echecs de retirer les pièces les plus fortes de l'adversaire, affaiblissant ainsi sa position et peut également permettre de créer des opportunités et des ouvertures pour les pièces restantes, d'autre part la possibilité de s'évader pour une pièces prisonnière.

Capture simple : Une capture simple pour être définie comme tout déplacement d'une pièce qui aboutit sur une case occupée par une pièce d'une couleur opposée entraîne la capture de cette dernière (prisonnère), et on stocke ces captures dans des ensembles pour chaque type de pièce donné:

`void simple_jumps(struct game_t game , struct set* sj)`: pour les captures simples des pions.

`void simple_jumps_capture(struct game_t game , struct set* sjc)`: pour les captures dans le cas d'un saut simple.

`void multiple_jumps_capture(struct game_t game , struct set* mjc)`: pour les captures dans le cas d'un saut multiple.

`void semi_diagonal_jumps_capture(struct game_t game, struct set* sdjc)`: pour les captures dans le cas du mouvements semi_diagonal pour l'éléphant.

`void cardinal_translations_capture(struct game_t game, struct set* ctc)`: pour les captures dans le cas du mouvement cardinal du tour.

Une pièce capturée deveint prisonnière, pour gerer cette nouvelle implémentation on crée une structure $prisoner\_t$ qui décrit l'état de la pièce capturée et pour manipuler plusieurs prisonèrs on définie une structure $jail_t$ qui contient les différents prisonèrs des deux joueurs. les deux structures sont définie de la manière suivante:

```
struct prisoner_t{
  enum color_t c;
  enum sort_t s;
  unsigned int i;
};

struct jail_t{
  unsigned int len_white;
  unsigned int len_black;
  struct prisoner_t cells_white[WORLD_SIZE];
  struct prisoner_t cells_black[WORLD_SIZE];
};
```

Mais pour ajouter une nouvelle dimension au jeu on implémente la posibilité d'évasion.

Tentatives d'évasion : Si une pièce est capturée, elle a la possibilité de tenter de s'échapper.

Cependant, cette évasion n'est possible que si la case où elle a été capturée est vide. La réussite de cette évasion est aléatoire, avec une chance de 50% de réussir. Si l'évasion échoue, la pièce reste capturée et le plateau de jeu reste inchangé.

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.

5 Configuration de compilation et exécution

5.1 Makefile

Les fichier .c décrivent dans ce rapport ont été construit et exécuté à l'aide d'un Makefile. Ce fichier de configuration automatise les tâches de compilation et d'exécution, facilitant ainsi la maintenance et l'extensibilité du logiciel. Les options de compilation et les dépendances ont été définies dans le Makefile, ainsi que les cibles pour construire et exécuter le programme. Nous avons utilisé GNU Make, un outil populaire pour créer des Makefiles, pour créer le Makefile de ce projet. En cours de développement, nous avons rencontré des problèmes avec les dépendances de bibliothèques, qui ont été résolus en ajoutant des options de compilation supplémentaires dans le Makefile. La méthodologie de compilation et d'exécution décrite ici a permis un développement efficace et une exécution stable de tous les fichiers.

5.2 La manipulation des options de ligne de commande

Pour manipuler la ligne de commande on a utiliser *getopt*, c'est un moyen efficace pour la gestion des optiosn et des arguments lors de la compilation du projet, donc l'utilisation de getopt nous

a permis de rendre le terminal une interface utilisateur intuisive et personnalisable. Il utilise une bibliothèque standard *getopt* de C. Se qui nous a permis de définir avant l'éxusion du projet le type de victoire souhaité par l'utilisateur, le nombre de tours et le nombre qui génére *rand*().

La bibliothèque getopt en C utilise la fonction getopt() pour analyser les options et les arguments passés à un programme lors de son exécution. Cette fonction prend en entrée les arguments standard argc et argv du programme, ainsi qu'une chaîne de caractères optstring qui définit les options valides pour le programme.

La fonction getopt() analyse ensuite les arguments dans argv en utilisant les options définies dans optstring. Pour chaque option valide détectée, getopt() retourne le caractère correspondant. Si une option nécessite un argument, celui-ci est stocké dans la variable globale optarg.

Les options peuvent être spécifiées de différentes manières. Les options courtes sont des caractères simples précédés d'un tiret (par exemple: -t pour le type de victoire et -m pour le nombre de tours). Les options longues sont des chaînes de caractères précédées de deux tirets (par exemple: –option).

Après avoir analysé tous les arguments, getopt() retourne -1 pour indiquer qu'il n'y a plus d'options valides. Les développeurs peuvent utiliser cette valeur pour itérer sur les options et les arguments restants.

Il est important de noter que getopt() modifie l'ordre des éléments dans argv pour les options traitées. Les arguments restants sont décalés vers le début de la liste argv.

6 Références

7 Annexes

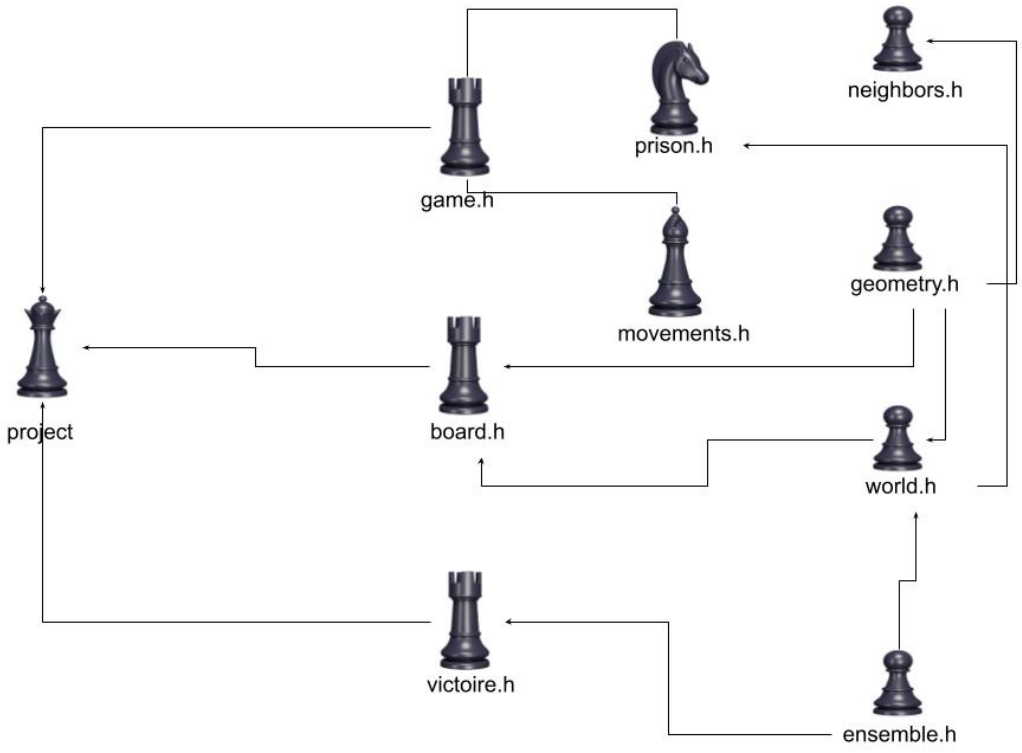


Figure 1: Graphe des dépendances