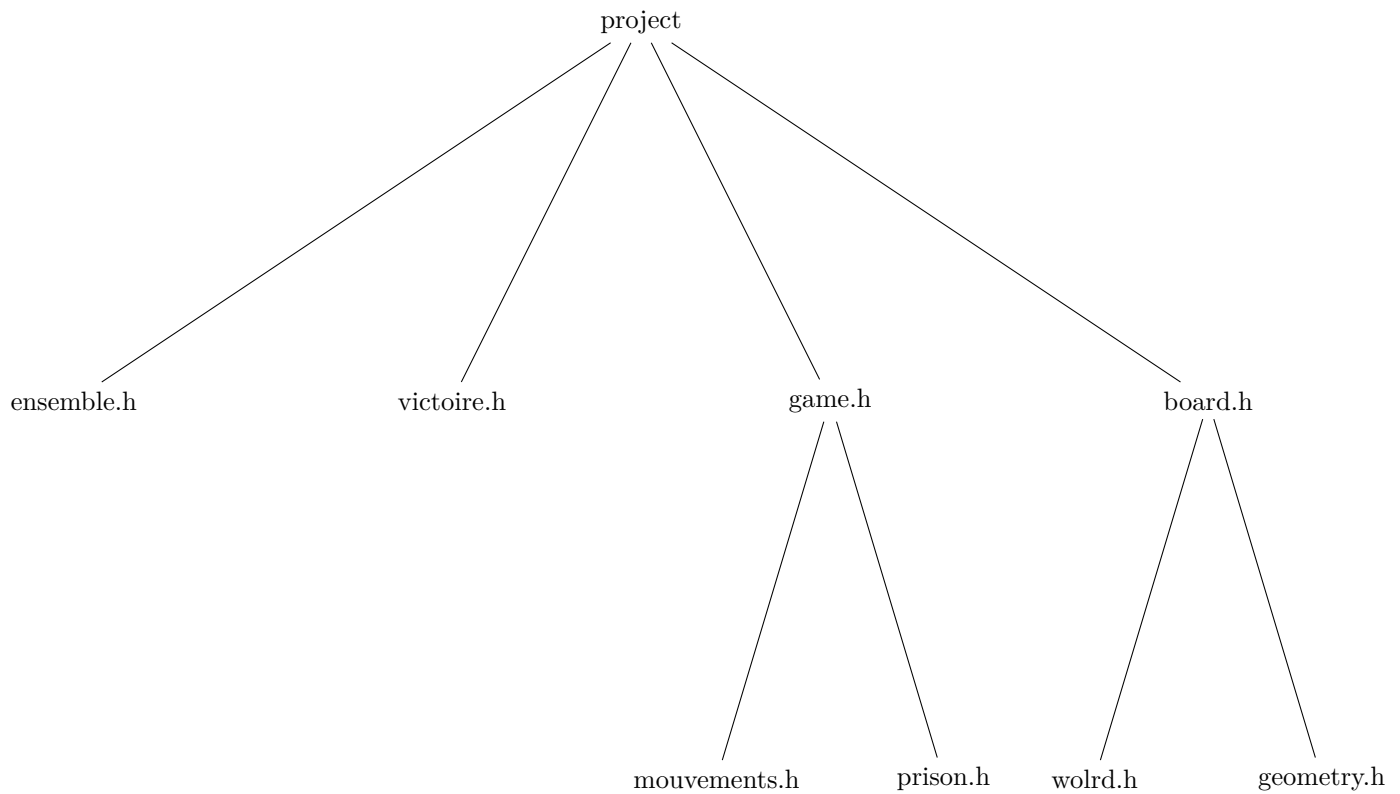


Enseirb Matmeca  
Rapport de projet S5  
MANSUBA

MOHAMMED BOUHAJA ET AMIRA ELOUAZZANI

January 9, 2023

Contents



## 1 INTRODUCTION

Mansuba est un jeu de plateau , ancêtre de Shtranj , qui a comme but mettre l'autre de joueur en situation de mat .

### 1.1 Problématique

Le but de notre projet sera de jouer une partie de jeu aléatoire, puis rendre l'algorithme de plus en plus flexible et général et l'orienter finalement vers la victoire.

## 2 Hypothèse et démarches de validation

Pour jouer une partie du jeu, 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'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 un enum dir . ( paste enum dir ici) La possibilité d'accès à ses voisins sera contrôlée par la relation de la partie et qui choisira des voisins ceux qui sont accessibles par mouvement direct.

## 3 Version initiale du jeu

Dans cette partie le jeu a une configuration qui n'est spécifique mais qui est plus simple a manipuler. Le jeu s'effectue sur une grille simple dans un monde a  $WORLDSize = WIDTH \times HEIGHT$  case , que des pions simples enum sort pour les joueurs ayant la couleur blanc et noir enum color . Cette configuration est surtout gouvernée par geometry.h .

Pour jouer une partie aléatoire on aura besoin de :

### 3.1 Configurer le plateau de jeu

#### 3.1.1 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 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.

#### 3.1.2 Partie relation

La structure Neighbors prédefinira les voisins de chaque indice donné . Les voisins seront un tableau dont le contenu pour un indice donné est un tableau de vecteur ( 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 get\_neighbors qui cherche dans la structure

voisins l'élément d'indice rentré en paramètre et `get_neighbor` 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

## 3.2 Ensemble

Cette structure ( structure ensemble) 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'ensemble 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

## 3.3 Mouvements

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

Déplacements simples : pour les relever il suffit d'utiliser le `get_neighbors` pour un indice donné . Ils seront stockés par la fonction (`deplacement_simples`)

Saut simple : si le voisin dans une direction `j` est occupé, équivalent d'avoir le `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 (`saut_simple`)

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 ensemble donné qu'on a noté `place_visited` qui renvoie un 0 si la place n'est pas encore mentionnée dans l'ensemble qui sera notre condition pour arrêter la recherche. Ils seront stockés par la fonction de complexité élevé ( cite sa complexité ).

Ses fonctions prennent comme paramètre additionnel un ensemble. Il servira comme espace de stockage pour chacune des fonctions au lieu de retourner l'ensemble à la fin de chacune . En plus, la fonction `mouvement_disponibles` fera appel à toutes les fonctions de mouvement citée au-dessus et l'ensemble 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 ensemble.

## 3.4 Positions de départ et victoires

Le monde étant inaccessible par d'autre document autre que `world.h`. On a utilisé les fonctions `wold_set` et `world_set_sort` pour donner à un monde initialisé vide des positions pour chacun des pions. Le nombre des pions étant `HEIGHT`.

Il existe deux types de victoires. La victoire 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 victoire, le choix du type étant aléatoire pendant la partie, ou si on atteint `MAX_TURNS`.

## 3.5 Boucle de jeu

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 `choose_random_piece` qui retournera un indice. La fonction `choose_random_move` va chercher entre les mouvements disponibles pour cet indice et va ensuite choisir un indice auquel le pion va bouger. La fonction `move_piece` 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.