Projet Mansuba

Louis Peyrondet, Samuel Khalifa

EI5PR103 - Projet d'algorithmique et de programmation n°1
Responsable : David Renault
Département Informatique
S5 - Année 2022/2023

Contents

1 Présentation

Le projet de cette année, Mansuba, était la réalisation d'un *framework* pour la création de jeux de plateau. Le développement du projet s'est déroulé entre le 8 novembre 2022 et le 13 janvier 2023.

Une base immutable était fournie. Nous ne pouvions modifier les fichiers `geometry.h` `world.h` et `neighbors.h`. Toutes modifications dans `world.c` étaient écrasées. Le projet était divisé en *achievements*, des jalons de fonctionnalités. Les *achievements* étaient à implémenter dans l'ordre. Il y en avait 7, mais nous n'en avons implémenté que 6.

0. La base du projet. On devait implémenter un monde, des relations entre les positions du monde, le déplacement simple d'un pion, le saut simple d'un pion, le saut multiple d'un pion, la condition de victoire simple, la condition de victoire complexe, et mettre en place une boucle de jeu robot vs robot.

1. Implémentation de nouvelles pièces. On devait implémenter d'autres types de pièces, qui ont leurs mouvements à elles. On devait implémenter la tour et l'éléphant.

2. Implémentation d'un système générique de relations. On devait implémenter 3 relations différentes, la grille classique, la grille triangulaire, et une de notre choix. Nous avons opté pour une hexagonale.

3. Implémentation des captures et des évasions. Maintenant, une pièce peut capturer une autre pièce en se déplaçant sur sa position. Capturer une pièce interrompt le mouvement. Toute pièce capturée peut tenter de s'évader si sa position est libérée. Elle a 50% de chance de s'évader.

4. Implémentation de robots un peu plus intelligents. Jusqu'ici, nos robots jouaient de manière aléatoire. Maintenant, les robots doivent choisir le coup qui amène la pièce choisie à la position la plus proche d'une des positions de départ du joueur adverse

5. Implémentation d'un objet générique de configuration. Il permet de définir les pièces autorisées, le type du plateau de jeu, les déplacements autorisés pour chaque pièce et si on peut capturer d'autres pièces ou non.

2 Choix techniques

Durant ce projet, nous avons copieusement abusé des mallocs. Cela pose certains problèmes de gestion de la mémoire (principalement des *memory leaks*) détectables avec `valgrind`, mais cela nous avantage grandement. En effet, cela nous permet de créer des objets "persistents". Ils restent en mémoire jusqu'à la fin du programme ou libération de la mémoire. Cela nous donne l'option d'utiliser des structures de données génériques.

2.1 Modularité

Nous avons divisé notre projet en plusieurs modules, ce qui nous a permis d'avoir une meilleure organisation au niveau des fichiers ainsi que dans la logique des composants. Nos structures de données ne sont utilisables qu'à l'aide d'interfaces ce qui permet de changer l'implémention sans perturber les autres modules.

2.2 Structure du jeu

```
typedef struct {
    uint turn;
    uint max_turns;
    player_t *current_player;
    enum victory_type victory_type;
    struct world_t *world;
    array_list_t *captured_pieces_list;
    array_list_t *starting_position;
} game_t;
```

Notre structure game_t contient tous les éléments nécessaires pour le déroulement d'une partie du jeu. Avec le recul et l'avancement dans le projet, une amélioration que nous voulions mettre en place mais n'avons pas eu le temps était de remplacer les champs captured_pieces_list et starting_pos par des tableaux d'array_list. Les tableaux seraient de taille nombre_de_couleurs afin d'avoir une array_list pour chaque couleur ce qui permet une séparation entre les pièces capturées selon leurs couleurs. Le bénéfice d'un tel changement est d'éviter les parcours inutiles, par exemple, lors de la libération de pièces, nous devons dans un premier temps parcourir la liste des pièces capturées afin de récupérer seulement celles de la couleur du joueur alors qu'avec la nouvelle implémentation cette étape n'est pas nécessaire donc il y aurait un gain de temps et une simplification du code.

2.3 Implémentation des structures de données

Premièrement toutes nos structures de données (sauf l'arbre par manque de temps) n'exportent pas l'implémentation de la structure sous-jacente. Dans le `.h`, il n'y a que la déclaration du nom de la structure, pas son implémentation.

Cela nous permet de réaliser un couplage faible entre les différents fichiers utilisant ces structures de données.

2.3.1 Tableau dynamique

Pour le tableau dynamique, nous avons besoin des opérations standards sur une liste, à savoir l'ajout, la suppression, l'insertion, l'affectation, et la récupération. Comme c'est un tableau sous-jacent, l'insertion et la suppression seront en complexité en temps O(n). Il faut déplacer tous les éléments dans le tableau. Ensuite, la récupération et l'affectation seront en complexité en temps O(1). Enfin, l'ajout d'un élément à la fin du tableau sera en temps moyen O(log(n)). En effet, lorsqu'on dépasse la taille du tableau, on doit allouer un nouveau tableau pouvant contenir tous les éléments précédents et l'élément à ajouter. L'approche naïve serait d'allouer un nouveau tableau de taille n+1. Le problème, c'est que si on veut rajouter pas 1, mais 2 élements, on devra allouer 2 nouveaux tableaux, et donc recopier la liste complète 2 fois. L'approche un peu plus subtile (celle qu'on a choisie), c'est de doubler la taille du tableau. Cela évitera de devoir copier tous les éléments de la liste trop souvent. Au désavantage de gâcher un peu plus de mémoire.

2.3.2 Arbre

Pour notre arbre, nous avons choisi de l'implémenter avec une structure récursive. Un noeud possède une valeur, une liste d'enfants et un parent. Nous avons opté pour un double chaînage car nous en avions besoin pour remonter l'arbre lors du calcul des mouvements possibles[6]. Pour la liste des enfants, nous avons utilisé le tableau dynamique défini précédemment.

2.3.3 Liste chaînée

Nous avions aussi besoin d'une liste chaînée. En effet, contrairement au tableau dynamique, l'affectation et la récupération sont en complexité O(n). Mais, les opérations d'insertion et de suppression sont en O(1). En gardant des pointeurs vers le dernier et premier élément de la liste, cela nous permet d'avoir un type qui peut fonctionner comme une pile et une file en temps constant. Nous avions particulièrement besoin d'une file pour le parcours en largeur[1].

2.4 Structure des fichiers

Les fichiers sont organisés en modules qui sont un couple .c et .h .

2.4.1 Les modules élémentaires

Ces modules forment la base de la logique du programme. Leurs définitions étaient imposées mais l'implémentation était libre.

- geometry : Gère les types de base ainsi que les définitions du projet.

- world : Gère les interactions avec le monde du jeu tel que placer ou récupèrer des pièces.
- neighbors : Gère les fonctionnalités pour récupèrer les voisins d'une case.

Les modules *world* et *neighbors* dépendent de *geometry*.

2.4.2 Le module util

Nous avons décidé d'utiliser un module *util* dont la majorité des autres modules inclut. Ce module contient les interfaces système comme *stdio.h*, des définitions comme *limit.h*, des définitions de type que nous utilisons dans l'ensemble du code tel que *uint* pour représenter le type *unsigned int* ou des structures ainsi que des fonctions d'erreur. Nous n'avons pas eu le temps mais les fonctions d'erreur auraient pu avoir leur propre module *erreur* afin d'avoir une meilleur organisation.

Notre choix de l'utilisation de ce module est de pouvoir enrichir les fichiers élémentaire que nous ne pouvions pas modifier ainsi que d'avoir à seulement inclure un fichier en en-tête des autres.

2.4.3 Les modules

- Le module distance permet d'avoir la distance de chaque cases vers une position gagnante. Cela nous permet de calculer les meilleurs mouvements pour une pièce. Son implémentation est détaillée dans la section 3.2.

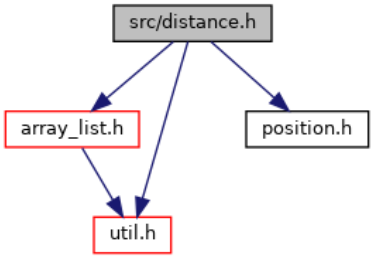


Figure 1: Dépendances du module distance

- Le module *configuration* permet de répondre à l'achievement 5, il encapsule toutes les fonctions liées aux paramétrages des configurations.

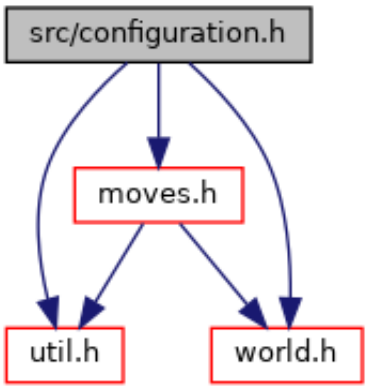


Figure 2: Dépendances du module configuration

- Le module moves contient la logique qui permet de récupèrer les mouvements possible pour les différents type de pièces. Toute l'implémentation est cachée dans le .c exposant seulement une fonction *get_moves* ce qui simplifie l'usage.

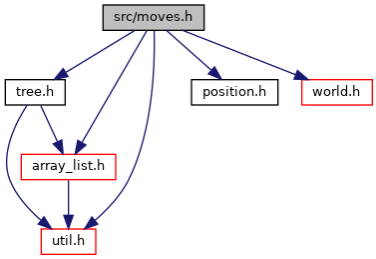


Figure 3: Dépendances du module moves

- Input est un petit module qui gère les entrées utilisateur sur le terminal. Il ne dépend de rien.
- Player sert à la gestion des joueurs dans le jeu. Sa seule dépendance est le module util.
- Le module player_handler permet l'intéractivité entre le jeu et un ou plusieur joueurs humains via le terminal.

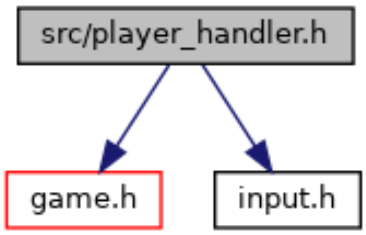


Figure 4: Dépendances du module player_handler

- Enfin le module game s'occupe de l'éxécution d'une partie du jeu.

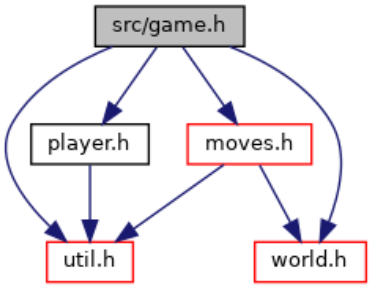


Figure 5: Dépendances du module game

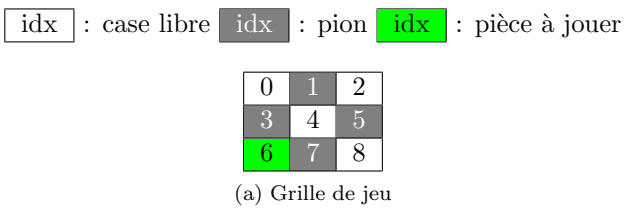
Tous ces modules sont inclus dans le fichier de plus haut niveau *projet* qui s'occupe d'initialiser le jeu puis de lancer la partie. Le graphique complet de dépendance de celui-ci est disponible en annexe, figure 10. Avec ce graphique on peut voir l'entièreté des dépendances depuis *projet*.

3 Principaux algorithmes

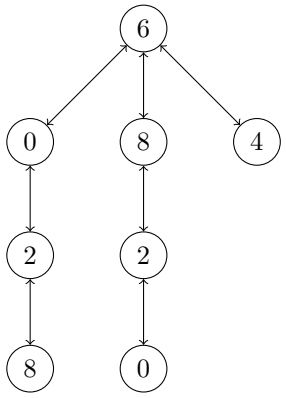
3.1 Mouvements des pièces

3.1.1 Pion

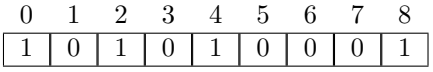
Pour le mouvement des pièces, nous avons choisi de le représenter comme un arbre des positions. Un noeud et ses parents représentent tout le chemin pour arriver à la position au noeud. Voici un exemple :



(a) Grille de jeu



(b) Arbre des mouvements possibles



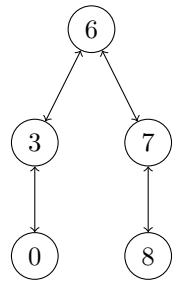
(c) Ensemble des mouvements possibles

Figure 6: Deux représentations des mouvements possibles pour un pion

Cet exemple nous montre 2 représentations des déplacements possibles pour le pion à l'indice 6 sans compter les captures. L'ensemble à quelques avantages comparé à l'arbre. Premièrement, pas besoin d'allocation dynamique, on peut juste faire un ensemble statiquement alloué qui a la taille du monde. Ensuite, on a une complexité O(1) pour voir si on n'est pas déjà allé sur une case lors de sauts multiples. L'implémentation en arbre, elle, doit être allouée dynamiquement, et il y a une complexité O(h) pour voir si nous ne sommes pas déjà allé sur une case (grâce au double chaînage). Mais, contrairement à l'implémentation avec un ensemble, nous pouvons savoir exactement quel chemin a été emprunté pour aller à une position. Prenons les sauts multiples pour arriver à la case 8. Dans le jeu des dames chinoises classique, il y aurait une stratégie : est-ce que je prends le pion à la case 7, ou ceux aux cases 1 et 5 ? Or, l'implémentation

en ensemble ne permet pas de définir quel chemin nous avons emprunté. Tandis qu'avec l'implémentation en arbre, il suffit de remonter l'arbre pour voir quel chemin nous avons emprunté. Même si l'arbre complexifie le problème, il permet de rendre les mouvements bien plus génériques.

3.1.2 Tour



(a) Arbre des mouvements possibles

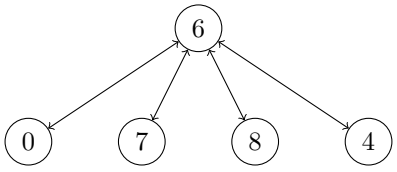
0	1	2
3	4	5
6	7	8

(b) Grille de jeu

Figure 7: Représentation des mouvements possibles pour une tour

On remarque que pour la tour, les chemins sont de longues branches. En fait, notre arbre représente les *dépendances* entre les mouvements. Pour aller à la case 0, la tour doit d'abord passer par la case 3. Cela est représenté dans notre arbre. Nous avons opté pour cette représentation pour la tour, car il y a une vraie notion de déplacement pour la tour. La tour ne saute pas à la fin.

3.1.3 Éléphant



(a) Arbre des mouvements possibles

0	1	2
3	4	5
6	7	8

(b) Grille de jeu

Figure 8: Représentation des mouvements possibles pour un éléphant

Pour l'éléphant, on remarque que tous ses mouvements sont à la même hauteur. En effet, contrairement à la tour, l'éléphant saute à la case, il n'y a pas de notion de déplacement entre la case 7 et la case 8 par exemple.

3.2 Choix des meilleurs mouvements

3.2.1 BFS

Pour pouvoir calculer les meilleurs mouvements, nous avons utilisé un Parcours en Largeur (Breadth First Search : BFS) sur le plateau. Premièrement on initialise la distance de toutes les cases avec une valeur maximale puis le principe est de créer une file dans laquelle on ajoute toutes les positions de départ d'une couleur. Ensuite tant que la file n'est pas vide, on récupère la position en tête, et pour chacun de ses voisins, si sa distance est inférieure on met à jour sa distance, qui est maintenant égale à la distance de notre position+1, puis on l'enfile. Ainsi chaque case se voit attribuer sa distance à la position de départ la plus proche pour une couleur.

Cela nous permet d'avoir la distance d'une case donnée vers la position de départ d'une couleur avec une certaine relation. Ensuite il suffit de changer la relation, vu que le module *neighbors* s'adapte à la relation, et/ou la couleur et de relancer l'algorithme.

Algorithm 1 BFS pour calculer les distances pour une couleur et une relation

Require: $C \in couleurs$
Require: $R \in relations$
Require: $positions\_de\_depart\_de\_C \in file$
Require: *Toutes les distances des autres positions initialisees au max*
  while $!estVide(file)$ **do**
    $position \leftarrow tete(file)$
    $voisins \leftarrow get\_neighbors(position)$
    for `voisin in voisins` **do**
      if $distance(position) + 1 < distance(voisin)$ **then**
        $set\_distance(distance(position) + 1, voisin, C, R)$
        $file \leftarrow ajout(file, voisin)$
      end if
    end for
    $file \leftarrow defiler(file)$
  end while

3.2.2 Table de correspondance

Ces distances sont enregistrées dans un tableau ce qui nous permet d'avoir une table de correspondance (*lookup table*), pour avoir en temps constant toutes les distances lors d'une partie vu que celle-ci est calculée lors de l'initialisation du programme. Concernant la complexité en espace, elle réserve taille_du_monde * nombre_de_couleurs * nombre_de_relations * taille_short_int octets. Pour un jeu standard de configuration HEIGHT=4 WIDTH=5, avec 2 joueurs et 3 relations différentes, la taille allouée est de 240 octets, ce qui est correct. Nous avons choisi d'utiliser des short int (2 octets) à la place des int (4 octets) pour réduire la taille totale allouée, le seul inconvéniant que cela ajoute est que la distance

maximale pouvant être enregistrée sans overflow est de *65 535* mais cela est largement suffisant pour nos utilisations.

Pour choisir le meilleur coup à jouer, on regarde parmi toutes les positions atteignables depuis notre case et on choisit celle dont la distance est la plus petite dans notre table de correspondance.

Nous avons opté pour une représentation de cette table par un tableau contigu en mémoire afin de favoriser le cache cpu lors du BFS avec la localité spatiale et temporelle des accès mémoire.

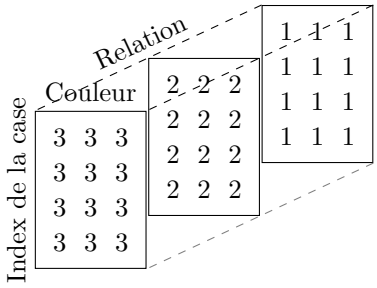


Figure 9: Représentation de la table de correspondance

4 Tests

Tous les tests du projet sont rassemblés dans le dossier tst/ . Un couple de fichier main.c et main.h rassemble tous les prototypes et exécute tous les tests présents dans les autres fichiers. Grâce à cette organisation lorsque nous ajoutons un module à notre projet, pour le tester, il suffit de créer un fichier .c du même nom, d'y ajouter les définitions des fonctions de tests puis d'ajouter les prototypes dans le main.h et finalement d'ajouter ces fonctions au main.c pour avoir ces tests intégrés à notre environement de test. Un simple *make test* permet l'exécution de tous les tests.

4.1 Structures de données

Pour chacune des structures de données que nous avons utilisées (arbres, array_list, liste chaînée), il existe un fichier de test correspondant. Cela nous permet de nous assurer que ces structures, qui forment la base de notre projet et de nos algorithmes, fonctionnent sans problèmes. Dans ces tests, nous nous assurons que les fonctions donnent le résultat attendu dans les cas courants ainsi que dans des cas limites tel que la suppression dans une liste vide.

Ces tests sont également examiné par *valgrind* afin d'être sûr qu'il n'y ait pas de fuite de mémoire.

4.2 Modules principaux

Pour pouvoir tester nos modules principaux, on met en place une situation spécifique puis nous appelons une des fonctions du module et on compare cette nouvelle situation après l'exécution à celle qui est attendue.

Certains modules n'ont cependant pas de test car difficilement réalisable. C'est le cas, par exemple, de notre module player_handler, qui s'occupe de gérer l'interaction d'un joueur à travers le terminal, vu que simuler les entrées n'est pas trivial et surtout que c'est un module de haut niveau donc presque aucun module ne dépend de lui.

4.3 Mode de débug

Avec l'utilisation des niveaux de verboses lors du jeu combiné avec le module d'interaction qui nous permet de jouer via le terminal, nous pouvons également découvrir des bugs lors de l'exécution du programme à l'aide du niveau de verbose 2 qui affiche beaucoup d'informations sur la partie en cours.

Cela ne remplace en aucun cas les tests unitaires qui s'assurent de la fonctionnalité des modules mais il y a quand même des avantages à pouvoir trouver des bugs grâce à cette interactivité que nous pouvons par la suite ajouter aux tests unitaires pour renforcer ces derniers.

5 Mots de fin

Pour finir ce rapport, nous aimerions parler de ce que nous avons appris, ce qui nous a plu dans ce projet, et ce qui nous a déplu.

5.1 Ce que nous avons appris

Nous n'avons pas tant appris sur le plan technique comparé à nos camarades, cependant, cela peut s'expliquer par notre parcours. En effet, l'un a fait une licence informatique, tandis que l'autre a fait un DUT Informatique. Nous avions donc déjà un fort bagage technique. Néanmoins, ce projet nous a tout de même permis de consolider nos connaissances, et même d'apprendre de toutes nouvelles choses. Par exemple, nous avons appris que pour déclarer une fonction sans paramètres, il fallait la déclarer comme ceci : `type nom(void);`. En effet, si nous ne mettons pas le `void`, cela n'est plus un prototype de fonction. Cela peut donc causer plusieurs soucis au niveau de la compilation (incohérences `.h` et `.c`). Pour détecter ce type d'erreur, nous avons donc appris qu'il fallait utiliser le tag gcc `-Wstrict-prototypes`. De plus, nous nous sommes efforcés d'utiliser des principes de programmation en C plutôt obscurs, tels que ne pas utiliser d'assert dans le code en production, car avec un tag gcc, on peut désactiver les assert. Nous avons donc défini une macro qui permet de vérifier si un malloc a bien alloué de la mémoire, sinon, le programme plante.

5.2 Les + du projet

On était relativement libre sur le projet. On pouvait implémenter une solution algorithmique différente de celle proposée, et nous avions un minimum à implémenter, mais pas un maximum. De plus, la seule structure de code vraiment imposée est celle de départ (même si cela nous a bien embêté). Aussi, nous étions bien encadrés, dès que nous avions une question, l'enseignant nous répondait de façon approfondie et compréhensible. De plus, les solutions algorithmiques proposées étaient intéressantes, voire intrigantes. Enfin, le sujet était intéressant dans son ensemble.

5.3 Les - du projet

Les fichiers `.c` et `.h` imposés étaient plutôt embêtants. Nous en avons déjà parlé plus haut [2.4.2], mais cela nous a forcé à faire un fichier `util.h` alors que nous aurions pu mettre certains types dans `neighbors.h` ou `geometry.h`. Ensuite, le fait qu'il n'y ait pas d'*Issue Board* sur la forge nous a forcé à créer un trello, ce n'est pas si grave, mais tout de même embêtant.

6 Annexe

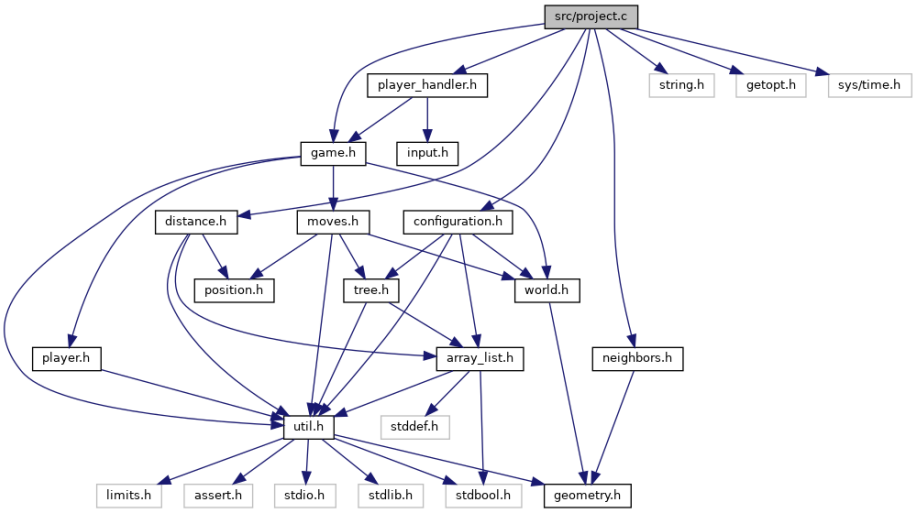


Figure 10: Le graphe de dépendances de notre fichier main