



Rapport de projet

Mansuba

Mohamed Rekik
Théodore Jankowiak

Projet de programmation
Département Informatique
S5 - Année 2022/2023

Table des matières

1	Introduction	2
1.1	Le projet	2
1.2	L'équipe	2
1.3	Les outils	2
1.4	Le jeu	2
2	Implémentation	5
2.1	Graphe de dépendances	5
2.2	Les bases du jeu	5
2.3	La modélisation du plateau	5
2.4	La modélisation des relations entre les cases	6
2.5	La modélisation des coups	6
2.6	La représentation visuelle du jeu	8
3	Les compétences développées	8
3.1	Programmation en C	8
3.2	Le travail en équipe	9
4	Bilan	9

1 Introduction

1.1 Le projet

L'objectif de ce projet est de coder dans le langage C un algorithme permettant de simuler des parties de [Mansuba](#), avec une possibilité de varier les règles, les formes de plateau et même les pièces utilisées.

Le travail est réparti en "achievements" numérotés de 0 à 7, un nouvel "achievement" étant débloqué lorsque l'une des équipes valide le précédent.

1.2 L'équipe

Notre équipe est composée de Mohamed Rekik et Théodore Jankowiak, nous sommes deux élèves en première année de la filière informatique à l'ENSEIRB-MATMECA.

Initialement, nous devions être encadrés pour ce projet par Monsieur Khamari mais ce dernier n'a pu nous assister pour des raisons de santé, et c'est finalement Monsieur Allali qui nous a pris en charge. Tout au long du projet, le directeur du module et auteur du projet Monsieur Renault a également été à notre disposition pour répondre à nos questions.

1.3 Les outils

Notre projet est entièrement codé en C à l'aide de l'application Visual Studio Code et la coordination du travail à deux a été possible grâce à un dépôt Git.

Nous avons à notre disposition une documentation fournie concernant l'utilisation de Git et sur les attendus du projet.

Au commencement du projet nous ont été fournis trois fichiers de type "header" contenant certaines constantes du sujet et les déclarations de fonctions que nous devons implémenter: `geometry.h`, `world.h` et `neighbors.h`. Nous n'avons eu le droit de modifier que `geometry.h`.

1.4 Le jeu

Notre jeu comporte 6 versions différentes, une version étant définie par un type de plateau et une règle.

Il y a trois types de plateaux.

Sur le plateau rectangulaire, les voisins d'une case sont les huit cases qui lui sont frontalières et les positions initiales se trouvent sur la première colonne pour le premier joueur et la dernière colonne pour le deuxième.



Figure 1: Le plateau rectangulaire en début de partie.

Sur le plateau en étoile, une case a des voisins le long des deux directions cardinales et sur la première diagonale, et les positions initiales se trouvent sur les branches de l'étoile, chaque joueur a

ses positions initiales sur trois branches consécutives de l'étoile.



Figure 2: Le plateau en étoile en début de partie.

Sur le plateau cylindrique, on se retrouve dans le cas du plateau rectangulaire en ajoutant la possibilité de passer de la première ligne à la dernière et inversement.

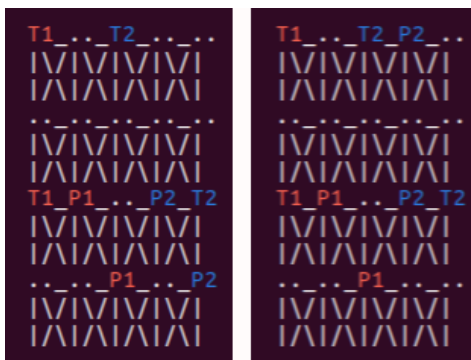


Figure 3: Le joueur 2 déplace un pion de la dernière ligne à la première selon la direction sud-ouest.

Il est possible de jouer sur ces plateaux avec deux règles différentes. Pour la règle simple, un joueur doit placer une de ses pièces sur l'une des cases de départ de l'adversaire pour gagner la partie.



Figure 4: Fin d'une partie simple sur un plateau en étoile.

Pour la règle complexe, un joueur doit placer toutes ses pièces sur les cases de départ de l'adversaire pour gagner la partie.

Dans les deux cas, s'il y a dépassement du nombre maximum de tours autorisés (nous l'avons défini arbitrairement à $2 \times$ taille du monde), il y a égalité.

Le choix de la partie est fait au moment de l'exécution de la commande make en précisant la valeur de la constante SEED.

valeur de SEED	type de partie
0	carré simple
1	carré complexe
2	étoile simple
3	étoile complexe
4	cylindre simple
5	cylindre complexe

Figure 5: Tableau des valeurs de SEED.

Il est également possible de choisir à l'exécution la largeur et la hauteur du plateau; un choix de dimensions inadaptées pour un plateau triangulaire renverra un message d'erreur.

Chaque joueur peut utiliser trois types de pièces:

- Le pion: il se translate par une seule case ou effectue une succession de sauts sur d'autres pièces selon les deux directions cardinales ou même selon les deux diagonales.

	x	x	x	
	x	P	x	
	x	x	x	

x				x
T			T	
x	E	P		

Figure 6: Les types de déplacement possibles pour un pion.

- L'éléphant: il peut accéder à quatre cases : selon chaque diagonale en sautant une case.

x				x
		E		
x				x

Figure 7: les déplacements possibles pour un éléphant.

- La tour: elle peut se déplacer d'autant de case qu'on le souhaite le long des deux directions cardinales tant qu'elle ne rencontre pas d'autre pièce.

		x		
		x		
x	x	T	x	x
		x		
		x		

Figure 8: Les déplacements possibles pour une tour.

Notre programme génère des parties automatiques et pseudo-aléatoires. Le choix des pièces de la partie, le choix du joueur ouvrant la partie, le choix de la pièce jouée à chaque tour et le choix du coup joué avec cette pièces sont choisis pseudo-aléatoirement. L'état du plateau et le coup joué à chaque tour sont affichés, puis le nom du gagnant en fin de partie s'il y en a un et le nombre de tours dont il a eu besoin pour gagner, ou la mention d'une égalité.

2 Implémentation

2.1 Graphe de dépendances

Pour utiliser dans un fichier des constantes et des fonctions déclarées dans d'autres fichiers on a besoin d'inclure les "headers" de ces autres fichier. Ces inclusions ne doivent pas être redondantes. Ces dépendances sont observables sur la Figure 9.

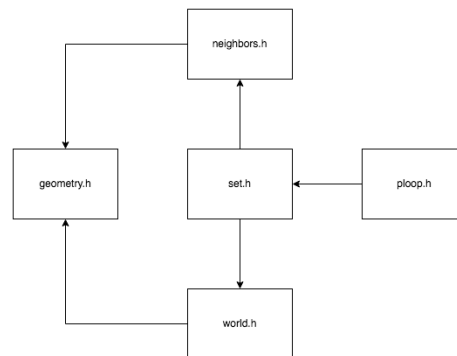


Figure 9: Le graphe des dépendances.
Une flèche de A vers B signifie A inclut B.

2.2 Les bases du jeu

Le fichier header de *geometry* fourni par l’auteur du sujet contient d’abord des constantes essentielles du jeu : hauteur et largeur du plateau que l’on peut modifier à l’exécution, taille du plateau qui découle des deux constantes précédentes.

Le fichier déclare également les couleurs correspondant aux joueurs, les états possibles de cases et les directions prises en compte dans le jeu (les deux directions cardinales auxquelles on ajoute les deux directions intermédiaires ”diagonales”).

Enfin, deux fonctions y sont déclarées, nous avons dû les implémenter dans le fichier .c correspondant mais nous ne les avons pas utilisées. Il s’agit de fonctions permettant de traduire par une chaîne de caractères les états de cases (occupation et couleur éventuelle) ainsi que les directions.

2.3 La modélisation du plateau

Dans notre jeu, le plateau est modélisé par un tableau de la taille de plateau désirée, chaque emplacement du tableau correspondant à une case du plateau caractérisée par son état et sa couleur. Les cases du plateau sont indexées de 0 à (taille du plateau - 1), de gauche à droite et de haut en bas.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Figure 10: L’indexation des positions d’un plateau rectangulaire pour WIDTH=5 et HEIGHT=4.

Le fichier header de *world* fourni par l’auteur du sujet déclare la structure de représentation du plateau que nous avons la liberté de choisir ainsi qu’une fonction d’initialisation de ce plateau et des fonctions de modification d’état et de couleur d’une case, nous avons implémenté ces fonctions.

Voici les structures utilisées pour modéliser le plateau :

```
struct space
{
    enum color_t color;
    enum sort_t sort;
};

struct world_t
{
    struct space board[WORLD_SIZE];
};
```

2.4 La modélisation des relations entre les cases

Pour modéliser les relations entre les cases, notre modèle utilise une table des relations, de la même taille que le plateau et dont chaque emplacement correspond à une case du plateau. Chacun de ces emplacements contient un tableau de voisins, un voisin étant défini par son index et par une direction. La définition de ces voisins dépend du type de plateau et du type de partie.

5	6	7		
10	11	12		
15	16	17		

Figure 11: Les voisins de la onzième case sur un plateau rectangulaire.

Le fichier header de *neighbors* fourni par l’auteur du sujet définit deux constantes : le nombre maximal de relations qui vaut 1 (nous n’avons pas trouvé d’utilité à cette constante) et le nombre maximal de voisins pour une case qui vaut 8. Il définit également une structure de vecteur, composée d’un index de case et d’une direction afin de représenter un voisin, ainsi qu’une structure de tableau de vecteurs qui permet de représenter une liste de voisins pour une case donnée. Enfin, y sont déclarées une fonction d’initialisation de la table des relations prenant en paramètre le type de partie, et deux fonctions permettant de récupérer l’ensemble des voisins d’une case donnée ou le voisin de cette case dans une direction donnée s’il existe.

Voici les structures utilisées pour modéliser la table de voisinage :

```
struct vector_t
{
    unsigned int i;
    enum dir d;
};

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

struct links_t
{
    struct neighbors_t neighbors[WORLD_SIZE];
};
```

2.5 La modélisation des coups

Dans notre jeu, les coups ne sont pas modélisés de la même façon selon l’emplacement dans le code. Pour les fonctions de recherche des coups possibles à partir d’une pièce, on utilise une structure associant une liste de positions d’arrivée et le nombre de ces positions d’arrivée. Dans la boucle principale, on a besoin qu’un coup porte l’information de la case de départ et de la case d’arrivée, et on utilise une nouvelle structure de binôme d’index.

Voici les structures utilisées pour modéliser les coups :

```
struct set_t
{
    unsigned int positions[WORLD_SIZE];
    unsigned int nb_positions;
};
```



```

struct move_t
{
    unsigned int s;
    unsigned int e;
};

```

2.6 La représentation visuelle du jeu

Pour bien suivre l'enchaînement d'une partie on a implémenté une fonction qui représente le plateau au moment où on l'appelle.

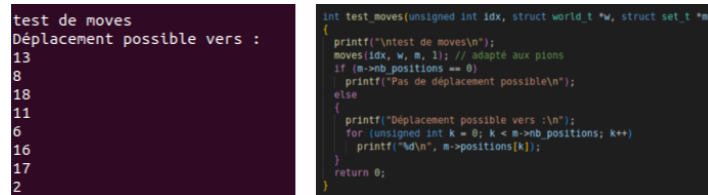
En effet on représente les positions vides par "...", les relations possibles entre deux positions par des traits et les pièces par la première lettre de leurs noms, le numéro du joueur qui les possède et on colore tous les pièces d'un joueur avec la même couleur (le rouge pour le premier joueur et le bleu pour le deuxième joueur).

3 Les compétences développées

3.1 Programmation en C

La programmation en C n'étant pas au programme des classes préparatoires, nous avons tous les deux découvert ce langage au début de notre première année à l'ENSEIRB-MATMECA. Au début de ce projet, nous avons terminé le premier module d'enseignement de la programmation impérative, mais nous avons encore de nombreuses fragilités de débutants en C. Il est donc normal que ce projet nous ait fait progresser dans ce domaine, et cela a notamment été possible grâce à la présence de nos encadrants.

Pendant le premier tiers temporel de la réalisation de notre projet, nous ne faisons que des tests pour les dernières fonctions implémentées, tests que nous ne sauvegardions pas. Cette méthode posait problème car les dernières implémentations peuvent avoir des conséquences sur la validité d'anciennes fonctions. De plus, nous ne faisons pas assez régulièrement de tests, et cela impliquait l'apparition d'une grande quantité d'erreur à la compilation qui prenaient plus de temps que nécessaire à régler. Nous avons donc rigoureusement construit un fichier de test qui à chaque exécution vérifie la quasi totalité des fonctionnalités de notre code et imprime les résultats.



```

test de moves
Déplacement possible vers :
13
8
18
11
6
16
17
2

```

```

int test_moves(unsigned int idx, struct world_t *w, struct set_t *m)
{
    printf("test de moves\n");
    moves(idx, w, m, 1); // adapté aux plans
    if (m->nb_positions == 0)
        printf("Pas de déplacement possible\n");
    else
    {
        printf("Déplacement possible vers :\n");
        for (unsigned int k = 0; k < m->nb_positions; k++)
            printf("%d\n", m->positions[k]);
    }
    return 0;
}

```

Figure 12: Un test de la fonction move.

L'élaboration de tests fait partie du métier de développeur, et constitue même parfois une spécialisation de cette profession. Il est évident que nos progrès dans ce domaine nous seront utiles dans la suite de notre parcours académique en informatique et même dans notre carrière professionnelle.

Une des principales difficultés rencontrées pendant l'implémentation a été l'introduction du pseudo-aléatoire dans la boucle principale du jeu. Nous n'avions jamais été confrontés à cette nécessité auparavant. Nous avons d'abord appris l'existence de la fonction `int rand()` de la bibliothèque `stdlib.h`. Seulement, lors de nos tests, cette fonction générait systématiquement la même série de nombres : pour utiliser cette fonction efficacement, il faut initialiser la suite de nombres aléatoires qui va être donnée en utilisant la fonction `void srand(unsigned int seed)`. Pour avoir une suite de nombres différente à chaque exécution, nous donnons en paramètre à `srand` le temps affiché par l'horloge `time(NULL)`, avec la fonction `time_t time(time_t * pTime)` de la bibliothèque `time.h`.

Concrètement, il nous fallait générer des nombres pseudo-aléatoirement dans un intervalle d'entiers donné. Avec notre méthode, la fonction `rand()` renvoie un entier quelconque entre 0 et la valeur `MAX_RANDOM` définie dans `stdlib.h`. Pour obtenir un entier dans l'intervalle voulu, par exemple `[a;b]` il suffit de prendre le reste de la division euclidienne du nombre obtenu avec `rand()` par `b`, puis de lui ajouter `a`.

Cette méthode d'introduction du pseudo-aléatoire nous sera certainement utile dans nos prochains projets.

3.2 Le travail en équipe

Ce projet était à faire en équipes de deux. Afin d'éviter au maximum les situations de conflit sur le dépôt Git et de ne jamais être en décalage l'un par rapport à l'autre, nous avons fait quasiment l'entièreté du travail sur le cas de base à deux, en travaillant sur la même machine et en alternant dans le rôle de rédaction. Cette façon de travailler a été très formatrice, elle nous a obligés à comparer nos idées de modélisation et à les défendre par l'argumentation. Il y avait toujours une discussion autour de la vue d'ensemble d'une implémentation avant de débiter la moindre rédaction, nous obligeant à aller au bout de notre raisonnement à chaque fois et évitant de nous engager dans des impasses. Cette réflexion peut avoir lieu même lorsque l'on code seul, sous la forme d'un débat interne, et est donc reproductible dans nos futurs projets.

Néanmoins, notre façon de faire avait pour principal défaut sa lenteur, et nous avons divisé le travail pour le reste du projet. Il est alors devenu essentiel de maîtriser la gestion des conflits sur le dépôt Git.

En effet, étant tous les deux novices dans le domaine de la programmation, ce projet a été l'occasion pour nous de découvrir et de nous familiariser avec un outil essentiel aux développeurs aujourd'hui : le dépôt Git. Nous sommes vite arrivés à un enchaînement de réflexes pour alimenter le dépôt :

- utiliser la commande `git status` afin d'avoir du recul sur les modifications que l'on vient d'effectuer
- utiliser la commande `git add` afin de sélectionner les fichiers voulus dans le dépôt
- utiliser la commande `git commit` avec `-m "nom du commit"` pour nommer la révision en question
- utiliser la commande `git push` afin de communiquer cette révision au dépôt

Lorsque nous travaillions en parallèles et que l'autre élève avait déjà communiqué ses modifications, l'étape du `push` n'était pas possible : il fallait alors suivre ces nouvelles étapes :

- utiliser `git pull` pour récupérer les modifications en question
- utiliser `git merge` pour fusionner nos modifications
- régler les conflits fichier par fichier
- communiquer le résultat de la manière habituelle, vue au-dessus

4 Bilan

Au terme de ce projet, nous avons complété le cas de base, l'achievement 1 et l'achievement 2. Initialement, l'auteur du sujet avait rédigé 7 achievements et 6 achievements ont été débloqués par le groupe le plus avancé. Bien que nous soyons satisfaits du rendu de notre projet, ce résultat montre néanmoins que nous avons été un peu lents. Cela est certes dû aux progrès que nous avons à faire en programmation en langage C et en utilisation de Git au début du projet, mais nous aurions pu gagner du temps en divisant plus tôt le travail.

Nous avons dans l'ensemble beaucoup progressé et sans aucun doute nous avons acquis des compétences qui nous rendront plus performants à l'avenir, pour nos futurs projets académiques et professionnels.