Rapport de Projet du Semestre 5

Projet Mansuba

13 janvier 2023

CATTARIN Antton
EL HABTI Badr

Encadrés par GUERMOUCHE Amina

Table des matières

Table des figures

1 Introduction

Dans le contexte du projet de programmation du semestre 5, il nous a été demandé de réaliser le projet Mansuba.

1.1 Description du sujet

Le sujet portait sur la réalisation d'un jeu de plateau, semblable à un jeu d'échec ou de dames chinoises. Pour ce faire, une version de base était proposée, permettant de générer une partie aléatoire entre deux joueurs. Chaque joueur possède le même nombre de pièces pouvant se déplacer différemment selon leur type. Pour gagner, un joueur doit occuper une seule ou toutes les positions initiales de son adversaire, selon le mode de jeu choisi. Ensuite, des contenus additionnels (ou *achievements*) ont été ajoutés au fur et à mesure des séances.

Nous devions programmer ce jeu en langage `C`, en découpant le projet en plusieurs fichiers *.c* avec leurs fichiers *.h* d'entête.

La compilation de tous ces fichiers devait générer un exécutable produisant une partie aléatoire. Nous pouvions passer des paramètres à la ligne de commande lançant l'exécutable afin de modifier le nombre de tours maximal ou le type de victoire considéré par exemple.

En dehors du jeu en lui-même, nous devions écrire un code nécessaire aux tests, afin de vérifier les programmes écrits.

1.2 Outils utilisés

Pour réaliser ce projet, nous avons utilisé plusieurs outils permettant de faciliter le développement et la collaboration notamment.

1.2.1 Git et Thor

Bien qu'il n'ait pas été évident de se familiariser avec ses outils durant les premières séances (plus de détails sur ce point dans la partie 5.1.2), Git s'est montré extrêmement utile une fois bien maîtrisé. En effet, avoir accès à nos fichiers partout, et avoir une mise à jour automatique des modifications des autres membres du projet, a été particulièrement bénéfique.

L'interface de Thor nous a également aidé, notamment grâce aux tests qui étaient directement présents sur la forge. Ils nous ont permis de faire les premières vérifications sur nos modifications. L'utilitaire Git Inspector, également disponible sur la forge, nous a indiqué les complexités cyclomatiques de nos programmes. D'après le site internet Mobiskill [1], plus la complexité cyclomatique d'un programme est faible, plus il est facile à comprendre, et moins il y a de chemins d'exécution possible dans le fichier. C'est pourquoi il nous semblait important de limiter au maximum cette complexité. Elle peut être calculée à la main, mais Git Inspector le fait automatiquement.

1.2.2 Make

La compilation d'un grand nombre de fichiers peut s'avérer fastidieuse, tant la ligne de compilation peut être longue, ou le nombre de commandes important. Pour palier ce problème, nous avions à notre disposition l'outil Make afin d'automatiser cette compilation. C'est un outil permettant de compiler un ou plusieurs fichiers selon un paramètre (ou règle, que l'on retrouve dans le Makefile) passé à la commande *make*.

Le Makefile était déjà pré-rempli et nous l'avons adapté selon les fichiers que nous avons ajoutés au projet. Pour ce faire, nous avons créé une variable contentant tous les noms de fichiers *.o* à compiler. Ainsi, la compilation peut se faire en une commande (et non une commande pour chaque fichier) et en utilisant le moins de caractères possibles.

La compréhension et l'optimisation du Makefile n'a pas été évidente au début, mais lorsque nous l'avions correctement assimilé, cet outil s'est révélé très pratique et nous a fait gagner un temps considérable.

2 Conception globale

Comme vu dans la partie 1.1, notre projet est composé de plusieurs fichiers (ou modules) *.c* (ils sont au nombre de 12 dans notre cas, sans compter les tests) et leurs *.h*. Chaque module contient des objets ou des fonctions portant sur un même thème. Ce dernier peut être identifié facilement grâce

1. Disponible à https ://mobiskill.fr/blog/conseils-emploi-tech/quest-ce-que-la-complexite-cyclomatique/

au nom parlant du fichier en question. Par exemple, le fichier `victory.c` contient les fonctions nécessaires à la détection de victoire. Tous ces modules utilisent des fonctions présentes dans d'autres. Dans ce cas, un fichier est dépendant d'un autre, et ces dépendances (que nous avons essayé de rendre optimales) nous permettent de construire le graphe de dépendances de notre projet.

2.1 Dépendances

L'ensemble de nos fichiers et des dépendances entre eux est visible dans la figure 1 ci-dessous :

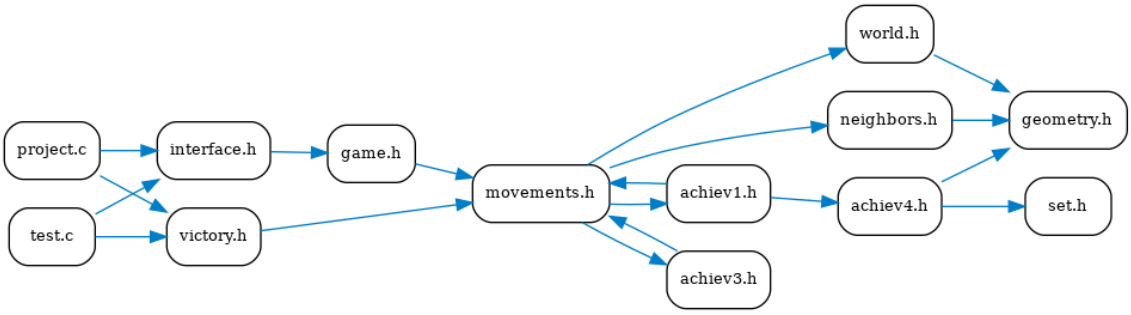

FIGURE 1 – Graphe de dépendances de notre projet

Grâce à ce graphique, on s'aperçoit clairement de la séparation entre les fichiers liés au déroulement d'une partie (`project.c, victory.h, interface.h, game.h`) et aux tests (`tests.c`) d'une part, et les autres contenant le code des objets utilisés pour le jeu et des fonctions agissant sur eux, d'autre part. Nous pouvons noter que ce graphe contient deux fichiers *.c*, car ils ne dépendent d'aucun autre. Il n'y a donc pas nécessité de créer leurs fichiers d'entête.

Nous avons décidé d'ajouter des fichiers additionnels pour chaque *achievement* supplémentaire. Nous avons fait cela pour nous y retrouver plus facilement entre les nouvelles fonctions et l'ancien code. Seul l'*achievement* 2 n'a pas de fichier dédié, car il était plus simple d'intégrer ses modifications dans des fonctions déjà existantes.

Ce graphique est celui avec le moins de relation que nous pouvions avoir, au vu des fichiers qui composent notre projet.

2.2 Structure du monde

Le jeu décrit dans la partie 1.1 se joue sur un plateau. Ce plateau peut être vu comme une matrice comportant `HEIGHT` lignes et `WIDTH` colonnes. Il a au total `HEIGHT` $\times$ `WIDTH` = `WORLD_SIZE` positions, numérotées de `0` à `WORLD_SIZE - 1`.

Nous avons choisi d'implémenter ce mode grâce à une structure `world_t` contenant deux tableaux (`c` et `s`) d'entiers, de taille `WORLD_SIZE`. Le premier regroupe les couleurs de chaque position (noir, blanc ou pas de couleur), tandis que la deuxième regroupe le type de pion (pion, tour, éléphant ou pas de pion).

Un monde 4×5 peut être vu comme suit :

c = 0 s = 0	... ...	... ...	... ...	... ...
... ...	... ...	... ...	... ...	... ...
... ...	... ...	... ...	... ...	... ...
... ...	... ...	... ...	... ...	... ...

FIGURE 2 – Exemple de représentation d'un monde 4×5

FIGURE 3 – Un monde tel que nous l'affichons lors du déroulement d'une partie

Ce monde est défini dans le fichier *world.c* sous la forme d'une variable globale interne à ce fichier. Pour y avoir accès depuis les autres fichiers, nous utilisons la fonction `world_init()` qui retourne un pointeur vers ce monde.

Les seules opérations agissant sur lui sont la modification et la récupération de la couleur ou du type de pion d'une position souhaitée. Ces opérations sont liées à l'utilisation d'un type que nous avons créé : les **ensembles**.

Un monde est de taille `HEIGHT`×`WIDTH`, il a donc une complexité quadratique, ici en $\mathcal{O}(HEIGHT \times WIDTH)$. Ce résultat nous a influencé dans notre choix d'implémentation décrit dans la partie suivante.

2.3 Ensembles et allocations dynamiques

Dans la version de base, il nous a été demandé de créer un nouveau type **ensemble**, nommé `set_t`, qui est ici un tableau de positions du monde, c'est-à-dire un tableau d'`unsigned int` (4 octets chacun). Pour cela, nous avons choisi de les définir via les allocations dynamiques.

2.3.1 Motivations

Afin de définir les ensembles, nous avions le choix entre une structure de taille fixe, avec un curseur repérant le dernier élément, ou une structure de taille variable. Pour faire notre choix, nous avons réfléchi aux situations dans lesquelles nous aurions eu besoin des ensembles.

Premièrement, nous avons les ensembles de pièces initiales, courantes et emprisonnées pour chaque joueur, qui, par définition, ont une taille qui ne peut pas dépasser `HEIGHT`. Dans ce cas-là, si la taille du monde augmente, on a une augmentation de la taille des ensemble qui est linéaire (car dépend uniquement de `HEIGHT` et non de `WIDTH`).

Enfin, les ensembles des mouvements possibles pour chaque pièce. Leur taille pourrait, dans le pire cas, atteindre `WORLD_SIZE`, si toutes les cases sont accessibles pour une pièce. Leur taille maximum dépend donc de ce paramètre, et, comme vu dans la partie 2.2, on aura une augmentation quadratique de la taille de ces ensembles.

Ils peuvent atteindre des tailles très grandes (surtout les ensembles des mouvements possibles) si les constantes `WIDTH` et `HEIGHT` augmentent. Par exemple, pour un monde de taille $100 \times 100 = 10000$, un ensemble des mouvements possibles aurait une taille de 40 ko. Ce cas n'arrivera pas dans

le cadre de ce projet (les parties seraient extrêmement longues), mais il est tout à fait envisageable de rencontrer un même type d'objet dans un autre domaine.

C'est pourquoi nous voulions absolument pouvoir définir ces ensembles à l'aide des allocations dynamiques de mémoire, afin qu'ils aient des tailles variables. De plus, cette façon de faire fut un très bon entraînement à l'utilisation de ces fonctions qui peut parfois être complexe.

Ce sont d'ailleurs ces difficultés qui nous ont posé problèmes, notamment dus aux fuites de mémoire. Nous reviendrons sur la résolution de ces problèmes dans la partie 5.1.1.

2.3.2 Mise en place

Ces ensembles sont définis comme étant le type `set_t` et possédant deux champs. Le champ `unsigned int *ptr`, obtenu lors de l'allocation dynamique de la mémoire, et le champ `int size`, taille de l'ensemble.

L'initialisation de ces ensembles se fait via une fonction qui va attribuer au champ `ptr` le pointeur obtenu avec la fonction `malloc()`. La taille de la mémoire allouée est égale à la taille d'un `int` (4 octets) multipliée par la taille passée en paramètre de la fonction.

En plus de cette définition, nos fonctions ont dû s'adapter à l'allocation dynamique. Par exemple, la fonction permettant d'ajouter un élément à un ensemble utilise `realloc()` afin de redéfinir la taille de la mémoire allouée :

```
void push_set(set_t *set, unsigned int x)
{
  resize_set(set, ((*set).size + 1));
  // on met x dans l'emplacement
  (*set).ptr[(*set).size - 1] = x;
}

void resize_set(set_t *set, int new_size)
{
  //on ajoute un emplacement de 4*(new_size - (*set).size) octets supplémentaires
  (*set).ptr = realloc((*set).ptr, new_size * sizeof(unsigned int));
  (*set).size = new_size;
}
```

Une autre fonction, absolument nécessaire et permettant de libérer la mémoire allouée et fixant la taille de l'ensemble à 0, est également présente.

Un certain nombre de fonctions, reliées aux mouvements des pièces ou aux victoires par exemple, utilisent les ensembles définis dans la partie 2.3.1. Pour éviter d'avoir à les passer en paramètres de chaque fonction, nous avons décidé de définir ces ensembles comme étant des variables globales.

2.4 Variables globales

Nous avions besoin d'utiliser des variables globales pour avoir accès à l'état du jeu, peu importe dans quelle fonction de quel fichier nous nous trouvions. Pour ce faire, nous avons défini comme variables globales les entiers `current_player`, qui indique le joueur actuellement en train de jouer, `achiev3` et `achiev4`, qui indique si l'*achievement* en question est actif ou non.

Comme dit précédemment, nous avons également des ensembles comme variables globales. Il y a les ensembles `black`(ou `white`)`_init`(ou `current`)`_set`, qui regroupent les ensembles des pièces initiales ou courantes de chaque joueur. Ils sont énormément utilisés dans les fonctions de mouvements (modification d'un élément, suppression lors de cas de capture...) ou de vérification de victoire (comparaison entre l'ensemble courant d'un joueur et l'ensemble initial de l'autre). On retrouve également `black`(ou `white`)`_prison` qui sont les ensembles regroupant les pièces emprisonnées pour chaque joueur.

3 Calcul des mouvements possibles

L'algorithme, sans doute le plus long et difficile que nous ayons à mettre à place pour ce projet, est celui du calcul des mouvements possibles d'une pièce. En effet, il y avait plusieurs situations à prendre en compte, telles le type de la pièce considérée et les positions des autres pièces dans le monde.

3.1 Contexte

La version actuelle de notre projet contient trois types de pièces, et donc trois fonctionnements différents du point de vue du mouvement.

Premièrement, les pions. Ils peuvent se déplacer d'une seule case dans les quatre directions cardinales (ici `SOUTH`, `NORTH`, `WEST` et `EAST`). Ceci correspond à un **déplacement simple**. Si la case voisine est occupée, il peut sauter par-dessus cette position et donc se déplacer de deux cases dans une même direction (sous réserve que le deuxième case ne soit pas occupée). Dans ce cas, on dit que le pion a réalisé un **mouvement simple**. Si une autre position voisine de la case d'arrivée est occupée, il peut de nouveau sauter et ainsi de suite. Ici, c'est un **saut multiple**. Voici une représentation des mouvements possibles pour un pion. Dans toute la suite, la position courante est celle en bleu foncé. Les mouvements possibles sont encadrés de cette même couleur. Les cases occupées par le joueur adverse sont de couleur bleu clair.

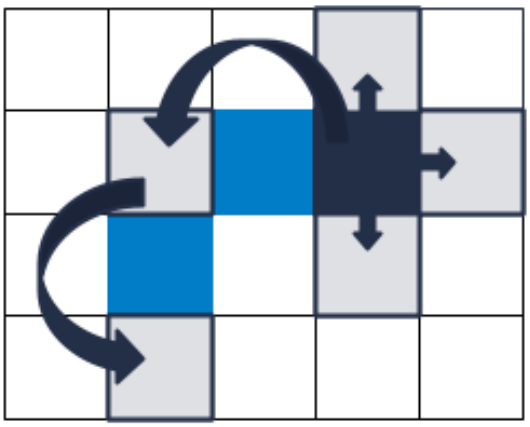

FIGURE 4 – Exemple de mouvements possibles pour un pion

Deuxièmement, les tours. Leur fonctionnement est identique aux tours des échecs que nous connaissons. C'est-à-dire qu'elles peuvent se déplacer de n'importe qu'elle distance dans les directions cardinales (les mêmes que celles des pions). Ci-dessous se trouve un schéma des mouvements possibles pour une tour.

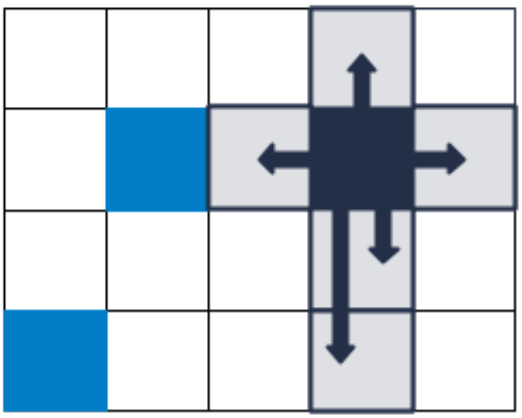

FIGURE 5 – Exemple de mouvements possibles pour une tour

Dernièrement, les éléphants. Ils peuvent se déplacer vers une position qui pourrait être atteinte avec deux mouvements consécutifs dans toutes les directions. Une représentation des mouvements possibles d'un éléphant est proposée ci-dessous :

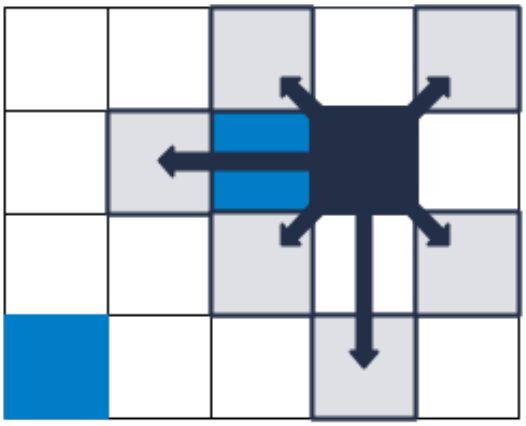

FIGURE 6 – Exemple de mouvements possibles pour un éléphant

À noter que les pièces peuvent **finir** leur mouvement sur une pièce adverse et ainsi la capturer (c'est-à-dire la sortir du jeu, avec une certaine probabilité de revenir sur la case où elle a été capturée si cette dernière est vide).

Nous allons nous intéresser dans la partie suivante aux mouvements des pions. Ceux des tours et des éléphants sont relativement simples.

3.2 Résolution

Notre solution contient deux fonctions. La première s'occupe de vérifier la possibilité de faire un déplacement simple ou un saut simple. Dans ce dernier cas, elle appelle notre deuxième fonction, auxiliaire récursive, qui va calculer tous les sauts multiples possibles. Dans la suite, nous noterons l'"ensemble des mouvements possibles" par l'"EMP", pour des raisons de praticité. Nous supposons également que l'*achievement* 3 est activé.

La première fonctionne de la manière suivante. Elle fait une itération sur les directions possibles d'un pion. Elle récupère la couleur du voisin dans chaque direction, et va faire différentes opérations selon le résultat.

Si la case n'a pas de couleur (c'est-à-dire vide), ou si elle contient une pièce adverse, on l'ajoute simplement à l'EMP.

Si elle appartient au joueur qui joue, on regarde la case voisine dans **deux** fois la direction. Si elle est occupée par le même joueur, on ne fait rien. Si la case est vide, on lance la fonction auxiliaire. Si elle contient une pièce adverse, on l'ajoute à l'EMP **sans** lancer la deuxième fonction.

Voici un exemple de fonctionnement de cette fonction, avec les cases regardées qui sont numérotées par ordre chronologique :

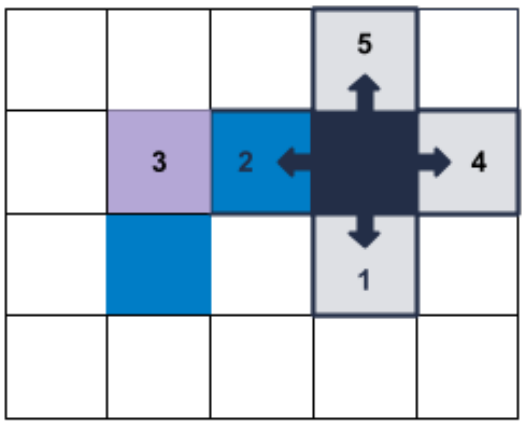

FIGURE 7 – Exemple de fonctionnement de la fonction calculant les mouvements possibles

La case violette est celle à partir de laquelle la fonction auxiliaire se lance.

La fonction récursive fonctionne un peu sur le même principe. Tout d'abord, si la case sur laquelle on se trouve est différente de la case de départ et qu'elle n'appartient pas déjà à l'EMP, on l'ajoute à cet ensemble. Ensuite, elle fait une itération sur les directions possibles et prend l'information sur le voisin dans chaque direction.

Si la case est vide, il ne se passe rien.

Si elle est occupée, on regarde le voisin dans deux fois la direction, comme précédemment. S'il contient une pièce du joueur actuel, la fonction ne fait rien. Si la case est vide, on l'ajoute à l'EMP et on rappelle la fonction auxiliaire **seulement** si elle n'appartient pas déjà à l'EMP. Cette condition empêche de faire une boucle infinie si on se retrouve dans une situation où une pièce pourrait faire un tour. Si on trouve une pièce adverse, on l'ajoute à l'EMP, toujours en vérifiant qu'elle n'est pas déjà dedans.

Ci-dessous sont schématisées les opérations faites dans l'ordre, des deux fonctions.

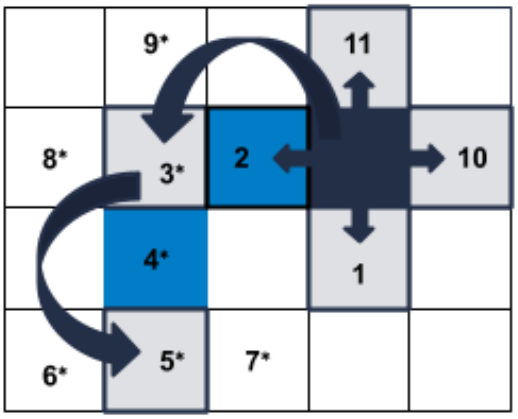

FIGURE 8 – Exemple de fonctionnement de la fonction auxiliaire calculant les mouvements possibles

Nous utilisons ici les mêmes codes couleur que dans la figure **??**. Les cases regardées par la fonction auxiliaire sont notées avec une *.

Chaque mouvement se terminant sur une pièce adverse supprime cette dernière de l'ensemble des pièces courantes du joueur en question et l'ajoute dans son ensemble prison. Nous avons eu une contrainte qui était que les pièces se déplacent uniquement en s'approchant le plus possible des positions de victoire. Pour cela, nous avons calculé l'EMP grâce aux fonctions décrites ci-dessus. Ensuite, nous avons appliqué à cet ensemble une fonction qui supprime tous les éléments, excepté ceux étant les plus proches des positions de victoire.

Ces algorithmes étant relativement longs et compliqués, il a été important de réaliser des tests solides pour les vérifier.

4 Les tests

Les tests des algorithmes forment une part importante d'un projet, pour s'assurer que le code écrit est bel et bien correct. C'est pourquoi il y a une nécessité de bien les formuler, pour couvrir l'ensemble des cas et s'assurer qu'il n'y ait aucun bug dans toutes les situations.

4.1 Nos tests

Pour réaliser nos tests, nous avons vérifié chaque fonction de notre projet une par une. Nous les avons exécutées dans des situations de jeu différentes ou en leur passant des paramètres différents. Ceci nous a permis de simuler l'ensemble des cas possibles.

Pour vérifier que les fonctions agissaient de manière appropriée, nous avons choisi d'afficher le résultat de chaque fonction, avec, à proximité, ce qui était attendu. Si les deux résultats sont différents, alors la fonction n'est pas correcte.

Voici, pour exemple, des tests réalisés pour la fonction `get_neighbor()`, qui retourne la position du voisin (s'il existe) d'une case dans une certaine direction, et `-1` sinon :

```
printf("Neighbor of %d in direction %s: %d ; expected -1\n",
       0, dir_to_string(2), get_neighbor(0, 2));

printf("Neighbor of %d in direction %s: %d ; expected 5\n",
       0, dir_to_string(-3), get_neighbor(0, -3));

printf("Neighbor of %d in direction %s: %d ; expected 6\n",
       12, dir_to_string(4), get_neighbor(12, 4));

printf("Neighbor of %d in direction %s: %d ; expected -1\n",
       14, dir_to_string(1), get_neighbor(14, 1));
```

Et voici ce qui est affiché dans notre cas :

```
Neighbor of 0 in direction NEAST: -1, expected: -1
Neighbor of 0 in direction SOUTH: 5, expected: 5
Neighbor of 12 in direction NWEST: 6, expected: 6
Neighbor of 14 in direction EAST: -1, expected: -1
```

Les tests sont validés ici, la fonction est donc correcte. Il faut noter que l'ensemble des situations possibles ne sont pas décrites ici. La validation d'autres fonctions utilisant `get_neighbor` participe également à ses tests (procédé valable aussi pour toutes les fonctions).

Par manque de temps, nous n'avons pas écrit de tests complets pour les *achievements* 3 et 4. Nous avons donc généré plusieurs parties aléatoires, analysé chaque coup joués pour repérer les problèmes éventuels.

Malgré nos tests, il y a d'autres erreurs que nous ne pouvions pas identifier. L'outil Valgrind nous a permis de les détecter.

4.2 Les tests Valgrind

Valgrind nous a été très utile tout au long du projet et nous avons appris à lire ses messages d'erreurs. En effet, nous en avons eu un nombre assez important. La grande majorité du temps, ces problèmes étaient dus à des fuites de mémoire, résultant en grande partie de la définition des **ensembles** avec les allocations dynamiques. Ces emplacements mémoires non libérés peuvent ralentir le programme s'ils sont en grand nombre. Et cela n'est pas en adéquation avec nos motivations à allouer dynamiquement la mémoire pour définir les ensemble. Nous verrons dans la partie 5.1.1 comment nous avons résolu ces dysfonctionnements.

Ce genre d'erreur n'est pas visible avec les tests que nous avions écrits. C'est pourquoi cet outil a été indispensable dans l'optimisation de la gestion de la mémoire et la réalisation de notre projet.

5 Problèmes rencontrés

Au cours de notre projet, nous avons fait face à plusieurs problèmes.

5.1 Les problèmes résolus

Parmi eux, nous en avons résolu un certain nombre.

5.1.1 Fuites de mémoire

Dans la partie 4.2, nous avons évoqué les problèmes de fuites de mémoire. Nous avons, fort heureusement, pu résoudre ces problèmes. En effet, pendant les premières séances ayant suivi l'implémentation des ensembles, nous n'avions pas le réflexe de libérer les emplacements mémoires via notre fonction `delete_set()`. Cela a mené à de nombreuses erreurs Valgrind, notamment visibles sur la forge. Pour remédier à ces problèmes, nous avons appris à appeler cette fonction aux bons endroits dans le code.

5.1.2 Prise en main de Git

Bien que Git soit un outil très pratique, nous avons mis un certain temps avant de nous familiariser avec lui.

Nous avons d'abord dû comprendre les différences entre les différents dépôts (local, distant). Ensuite, nous nous sommes habitués aux différentes commandes Git telles que *commit*, *pull*. Tout ceci s'est fait relativement rapidement. Cependant, la gestion des conflits a été une tâche bien plus complexe.

Dans un premier temps, pour ne pas avoir à faire face à ce problème, nous travaillions sur la même machine pour modifier un fichier. Cette méthode n'est évidemment pas efficace. En effet, nous avancions deux fois moins vite que si nous avions travaillé sur nos propres machines. Nous avons donc appris à gérer ces conflits et avons pu gagner en efficacité.

Malgré nos efforts pour corriger ces problèmes, certains sont restés irrésolus.

5.2 Les algorithmes inachevés

Du contenu pour notre jeu qui nous a particulièrement posé problème est l'implémentation de la grille triangulaire, demandée pour l'*achievement* 2.

Lorsque nous sommes passés à cet *achievement*, nous n'avions pas d'idée de comment implémenter cette relation et nous nous sommes directement rendus au suivant. C'est seulement récemment que nous avons trouvé une solution. Cependant, nous n'avons pas eu le temps de l'implémenter.

Dans une grille triangulaire, chaque position possède 6 voisins (exceptées celles sur les bords qui en ont moins). Pour réaliser l'implémentation, nous sommes partis d'une représentation de cette grille, et lui avons apporté des modifications. Le but était d'obtenir la grille classique, comme au départ du projet. Nous avons ensuite numéroté chaque voisin d'une position dans la grille triangulaire, et les avons retrouvés dans la grille classique. Il a été facile de remarquer une différence de modélisation entre les lignes d'indices pairs et impairs, comme détaillé dans les figures 9 et 10.

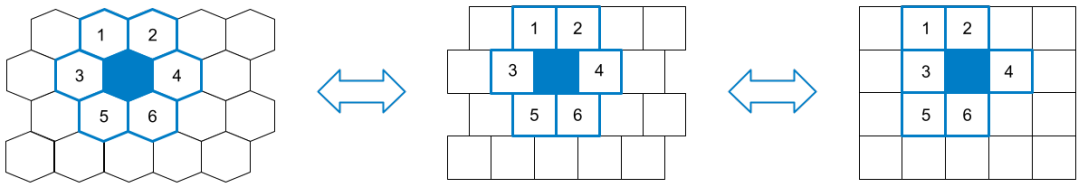

FIGURE 9 – Modélisation de la relation triangulaire pour les lignes d'indices pairs

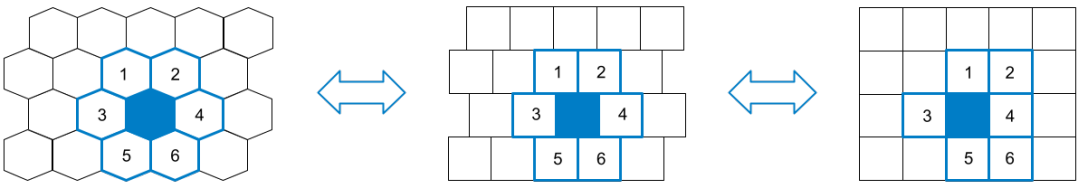

FIGURE 10 – Modélisation de la relation triangulaire pour les lignes d'indices impairs

Nous remarquons que la parité de l'indice de la ligne n'a pas d'incidence sur les positions 3 et 4. Les mouvements seront donc faciles à implémenter. Cependant, pour les autres positions, les difficultés apparaissent.

Prenons un exemple. Le voisin 1 d'une position se trouvant sur une ligne d'indice impair se trouve dans la direction `NORTH` d'après la représentation de la grille carrée. Or, ce même voisin pour une ligne d'indice pair se trouve dans la direction `NWEST`. Ceci signifie que dans le cas des sauts,

il ne suffit pas de regarder deux fois dans la même direction : il faut aussi prendre en compte la parité de l'indice de la ligne et ainsi adapter les directions.

Cette solution est probablement celle que nous aurions essayé de mettre en place avec du temps supplémentaire.

5.3 Les pistes d'améliorations

Avec plus de temps, nous aurions également pu améliorer plusieurs de nos façons de faire.

Premièrement, nos tests pourraient devenir problématiques si le projet venaient à s'agrandir. En effet, le fait d'afficher tous les résultats obtenus et attendus dans le terminal peut facilement devenir lourd, surtout s'il y a nécessité de les vérifier à la main. C'est pourquoi il peut être intéressant d'utiliser une méthode qui pourrait faire cela à notre place. Nous pourrions peut-être utiliser la fonction `assert()` du module `assert.h` et lui passer en paramètres : "$obtenu == attendu$". Si l'égalité est fausse, le programme s'arrête de lui-même en nous donnant la ligne et le fichier contenant l'erreur. Pour continuer sur ce point, nous pourrions également créer un fichier de tests par fichier contenant le code du projet. Ceci nous permettrait d'exécuter le programme de tests seulement pour le fichier contenant les fonctions souhaitées, et donc faciliter les vérifications.

Deuxièmement, nous aurions pu être plus rigoureux sur la réalisation de nos tests. En effet, à la place de les écrire après chaque implémentation fonction, nous les faisions par bloc afin de tout tester d'un coup. Cette méthode nous a coûté énormément de temps, puisque nous nous sommes plusieurs fois retrouvés à résoudre un gros problème plutôt que plusieurs petits. Ce sera un point sur lequel s'améliorer pour les prochains projets.

6 Conclusion

Ce projet nous a permis d'apprendre à travailler en groupe, bien que ce soit probablement plus difficile pour des projets à 5 personnes. Nous avons appris à nous répartir le travail dans le but de gagner en efficacité. L'outils Git nous a grandement aidé pour cela. Le fichier Makefile nous a également fait gagner énormément de temps grâce à la compilaton séparée.

Nous avons également amélioré nos compétences de programmation dans le langage `C`. Nous avons acquis des automatismes et avons appris à organiser notre code en plusieurs fichiers.