Projet de Programmation S6

2019/2020

Hex Project

Étudiants :
Nathan Decou
Aymeric Ferron
Aurélien Moinel
Saad Zoubairi

Encadrant :
M. Faverge

14 février 2021

1 Introduction

Hex Project est un projet de programmation impérative proposé aux élèves de la filière informatique durant le second semestre de leur première année. L'objectif de ce projet est l'élaboration d'un jeu et de joueurs automatisés en langage C. Au cours de cet exercice, les élèves sont amenés à mettre en pratique leurs connaissances en programmation et en algorithmique afin de résoudre les problèmes qui leurs sont posés.

1.1 Présentation du projet

Le jeu choisi pour l'année 2019/2020 est le *Hex*. Il s'agit d'un jeu de plateau à deux joueurs. Tour à tour, les joueurs vont assigner leur couleur à une case du plateau en cherchant à relier les deux bords qui leur appartiennent. Lorsqu'ils y arrivent, ils gagnent et la manche se termine.

1.2 Cadre de travail

La réalisation de ce projet s'est déroulée à distance en raison de la pandémie du COVID-19. Notre équipe, composée de quatre élèves, s'est réunie au moins une fois par semaine pendant quatre heures grâce à l'outil *Discord* pour avancer dans l'élaboration du jeu. La programmation en binôme (*pair-programming*) ayant été rendue difficile, nous avons veillé à communiquer régulièrement par échanges audios et textuels. Afin d'éviter les conflits sur le dépôt GIT, nous nous sommes répartis des tâches différentes et nous avons travaillé sur des fichiers distincts tout en nous entraidant. Nous avons adopté la méthode Agile Kanban afin de gagner en performance en nous appuyant sur le gestionnaire de projet en ligne `Taïga` (Annexe 6).

Chaque mardi après-midi, nous avons été encadrés par M. Faverge et M. Renault sur les outils *Slack* et *Discord*. Ainsi, nous avons pu avoir un retour régulier sur notre code ainsi que des réponses à nos questions.

1.3 Plan

Dans un premier temps, nous présenterons l'architecture logicielle de notre projet. Dans un second temps, nous discuterons nos choix d'implémentation et nous aborderons leur complexité. Enfin, nous présenterons nos tests, notre documentation et des pistes d'améliorations avant de conclure.

2 Projet

Dans cette partie nous présentons tout d'abord notre architecture logicielle puis nos choix d'implémentations avant de décrire nos tests, notre documentation et de proposer des idées d'amélioration.

2.1 Architecture logicielle

Après lecture du sujet, nous avons remarqué que le problème pouvait être traité selon une approche en programmation orientée objet (POO). Il a donc été possible de réaliser l'équivalent d'un **diagramme de classes** en l'adaptant aux fichiers sources et aux fichiers *headers*.

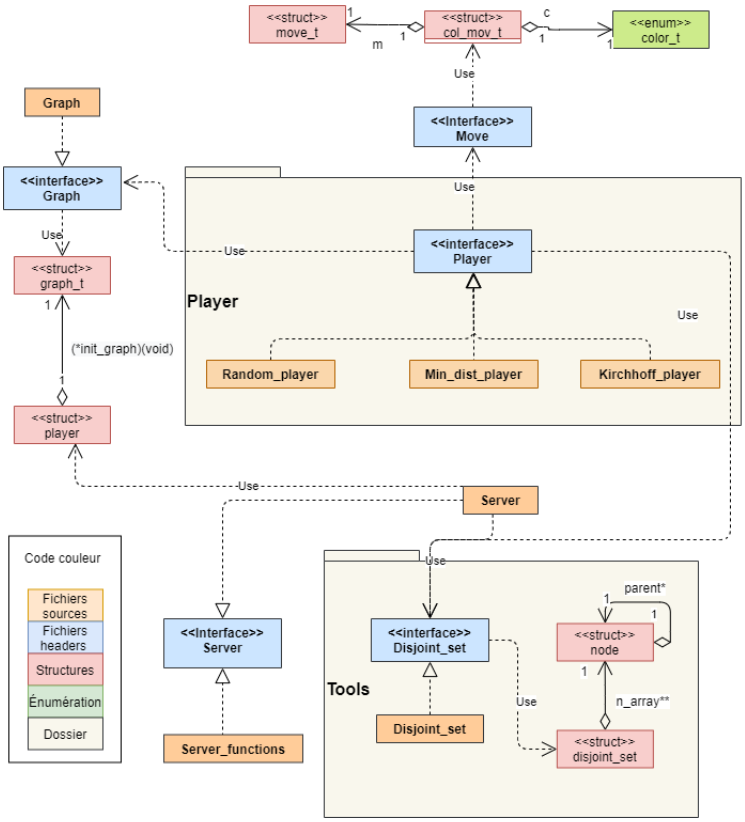

FIGURE 1 – Diagramme de classes simplifié de notre projet.

Nous avons réalisé une version simplifiée (Figure 1) reprenant les dépendances entre les différents fichiers afin de mettre en évidence l'ossature de notre architecture.

Nous avons également gardé à l'esprit qu'un projet informatique peut régulièrement être adapté par l'ajout ou le retrait de fonctionnalités. C'est pourquoi nous avons réalisé un diagramme plus complet (Annexe 7) mis à jour au fur et à mesure de l'avancée. En effet, toute modification est plus facile à réaliser si un schéma détermine la structure du programme.

Pour les besoins du dépôt central et des tests effectués par la Forge, nous avons réalisé un *Makefile* contenant plusieurs règles. Ces dernières permettent de compiler l'ensemble du projet (exécutable du serveur, bibliothèques des joueurs, documentation, tests, rapport) et de nettoyer le dépôt. Nous avons veiller à le rendre le plus convenable possible en gérant les dépendances entre les fichiers *.c* et leurs *headers*.

Trois types de règles ont été construites. Les premières vérifient si un fichier et ses dépendances ont été modifiées pour savoir s'il y a besoin de le recompiler. Les deuxième compilent les fichiers sources (*.c*) en fichiers objets (*.o*). Les dernières compilent les fichiers objets afin de les éditer entre eux pour créer les exécutables `server` et `alltests` ainsi que les bibliothèques de joueur (telle `librandom_player.so`). Notre client est `server.c`. Celui-ci utilise les bibliothèques (`graph.o`, `move.o`, `player.o` et `disjoint_set.o`) ainsi que des bibliothèques externes comme `gsl_spmatrix.h`.

Un fichier *README.md* permet de donner un aperçu de notre projet. Ce fichier sert également de page de garde à la documentation *Doxygen* qu'il est possible de générer avec la commande `make docs`.

Les fichiers qui composent notre projet sont répartis dans différents dossiers. Ainsi, à la racine on retrouve le *README.md*, le *Makefile* ainsi que le nom d'équipe. Le dossier **doc** contient le fichier *Doxyfile* qui permet la génération de la documentation. Le dossier **install** est vide la plupart du temps et est utilisé par la Forge pour procéder aux tests. Enfin, le dossier **src** contient toutes les sources rangées dans des sous-dossiers. Ainsi, les principaux fichiers du projet se trouvent à ce niveau tandis que les fichiers propres aux joueurs se trouvent dans le dossier **player** et que les tests se trouvent dans le dossier **tst**. Enfin, le dossier **tools** contient des outils permettant de gérer nos structures de données. Le dossier **in_developpement** contient les fonctionnalités que nous n'avons pas eu le temps de terminer.

Limiter les doubles inclusions de *headers* a été une priorité d'où la réalisation d'une conception en amont et l'ajout de **macro guard** pour rendre les *headers* idempotent. Ainsi, `server.c`, notre client, utilise de nombreux *headers* qui lui sont fournis à travers les inclusions successives dans `server.h`. Deux exemples de diagramme d'inclusions sont disponibles en annexe (Annexe 9 pour les inclusions du

fichier `alltests.c` et Annexe 10 pour les inclusions de `server.c`).

2.2 Implémentation

Cette partie présente la conception du jeu. Dans un premier temps, nous aborderons le fonctionnement du serveur permettant de faire jouer deux joueurs et dans un second temps, nous détaillerons l'implémentation de nos joueurs et leurs différentes stratégies.

2.2.1 Serveur

Le serveur représente l'exécutable principal de ce projet. Il gère le plateau et les interactions entre les deux joueurs. Le fichier `server.c` contient le `main` et par conséquent la boucle principale du jeu, tandis que le fichier `server_functions.c` possède l'ensemble des outils permettant de faire fonctionner cette boucle.

La première étape lors du lancement d'une partie est la création du plateau de jeu. C'est le serveur qui s'en charge. Cette option est personnalisable grâce à la spécification de paramètres lors de l'exécution du serveur dans le terminal. Les options sont recueillies à l'aide d'une fonction `parse_opts`. Ainsi, le jeu supporte trois formes de plateau : un plateau **carré**, un plateau **hexagonal** et un plateau **triangulaire** de taille variable. De la forme et de la taille vont dépendre le nombre de cases du plateau calculé dans une fonction dédiée.

La structure représentant le plateau se nomme `struct graph_t`. Dans la suite du sujet, on modélisera le plateau de jeu par un graphe dont chaque sommet représente une case du plateau. La structure `graph_t` possède trois champs distincts :

1. **num_vertices** (de type `size_t` qui garde en mémoire le nombre de sommets
2. **t** la matrice d'adjacence qui rend compte de la forme du plateau en gardant en mémoire la liste des voisins de chaque sommet. Cela permet de modéliser les coups possibles depuis une case. Il s'agit donc d'une matrice de taille `num_vertices * num_vertices`.
3. **o** la matrice arbitre de taille 2∗ `num_vertices` qui retient quels sont les sommets occupés par le joueur 1 et le joueur 2.

Puisque la forme du graphe dépend de la forme du plateau, trois fonctions différentes ont été développées, gérant les spécificités de chaque cas : quatre voisins par sommets pour la forme carrée, six voisins pour la forme hexagonale et trois voisins pour la forme triangulaire. Une fois la matrice d'adjacence `t` remplie, une fonction se

charge d'attribuer les quatre bords du plateau aux joueurs en initialisant la matrice `o`. La mémoire pour la structure du graphe est allouée dynamiquement car l'utilisation de la bibliothèque *GSL* l'oblige.

Une fois la génération du graphe terminée, le serveur doit charger les joueurs. Ces derniers sont des bibliothèques contenant un certain nombre de fonctions nécessaires pour le bon déroulement de la partie. Le serveur est muni d'une structure `struct player` qui va permettre de récupérer dynamiquement les fonctions propres à chaque joueur. Ainsi, cette structure contient sept pointeurs de fonctions. Au cours de l'initialisation, ces derniers vont pointer sur les fonctions des bibliothèques de façon à pouvoir être utilisées par la suite. On définit également la couleur du joueur (WHITE (1) ou BLACK (0)) et on initialise l'ensemble des sommets qu'il possède en lui fournissant une copie du graphe du serveur.

Afin de représenter l'ensemble des sommets possédés par un joueur, il a été décidé d'utiliser une **forêt**. Ce choix est intimement lié à notre fonction de terminaison de jeu, i.e. lorsque les deux bords du plateau sont connectés et que la partie se termine. Ainsi, le joueur voit le plateau sous la forme de plusieurs composantes connexes qui lui appartiennent. Ces composantes sont formées des sommets qu'il possède. Lorsque ces sommets sont voisins, ils font partie de la même composante connexe. La partie se termine lorsque les deux premières composantes connexes ne font plus qu'une, c'est-à-dire lorsque les deux bords du plateau sont fusionnées en une composante connexe unique, signe qu'un chemin a été tracé entre les deux zones de départ.

On commence par sauvegarder dans la structure du joueur un sommet faisant partie de la première composante connexe et un sommet faisant partie de la deuxième. A chaque tour de jeu, on gère l'ajout d'un sommet à l'ensemble de la façon suivante :

— si le sommet est adjacent à un sommet faisant déjà partie d'une composante connexe, on fusionne les deux composantes connexes.

— si ce n'est pas le cas, on crée une composante connexe indépendante.

Grâce à notre forêt, il est aisé de vérifier à chaque coup si le joueur a gagné ou non. En effet, une composante connexe est implémentée par un arbre dont la racine est un sommet et dont tous les fils sont les autres sommets de la même composante connexe. Ainsi, il suffit de vérifier si les deux sommets sauvegardés comme étant les deux extrémités du plateau ont la même racine pour vérifier s'ils font partie de la même composante connexe. Cette technique se nomme *Union-Find*.

Puisque cette méthode a été implémentée sous forme de **forêt**, il a été possible d'y ajouter des améliorations qu'il n'aurait pas été possible d'implémenter avec une liste chaînée. Premièrement la **fusion optimisée des racines** permet un gain de

temps lors de la fusion. En effet, l'arbre de plus petite taille est associé à l'arbre le plus grand, ce qui permet de réduire le nombre d'opérations nécessaire à la fusion. Ensuite, la **compression de chemin** est réalisée à chaque parcours de l'arbre, de façon à ce que tout noeud reliant la racine par un chemin soit relié directement à cette même racine. Ainsi, la complémentarité de ces deux améliorations permet d'atteindre une complexité en temps de $\mathcal{O}(\alpha(n,m))$, avec $\alpha(n,m)$ la réciproque de la fonction d'Ackermann. Nous fournissons ci-dessous la fonction d'Ackermann (équation 1) ainsi que sa réciproque (équation 2) :

$$\begin{cases} \mathcal{A}(0,n) & = n+1 \\ \mathcal{A}(m+1,0) & = \mathcal{A}(m,1) \\ \mathcal{A}(m+1,n+1) & = \mathcal{A}(m,\mathcal{A}(m+1,n)) \end{cases} \tag{1}$$

$$\alpha(m,n) = min\{i \geq 1 : \mathcal{A}(i, \lfloor \frac{m}{n} \rfloor) \geq log_2(n)\} \tag{2}$$

avec, m le nombre d'arêtes et n le nombre de sommets.
De ce fait la complexité en temps associée à cet algorithme est **quasi-constante**.

Le début de partie utilise la *règle du gâteau*. Pour initier la partie, un des deux joueurs propose un coup. Ce coup peut être validé ou refusé par le joueur adverse. S'il est validé, le coup appartient au joueur qui l'a joué, et la partie se poursuit. S'il est refusé, le joueur qui refuse vole le coup de son adversaire, et c'est à l'adversaire de rejouer. La partie se poursuit ensuite normalement, un coup après l'autre, tant qu'il reste des sommets libres ou tant qu'aucun joueur n'a gagné. On détecte quel est le joueur courant en comptant le nombre de tour : lors des tours pairs, le joueur 1 joue, lors des tours impairs, c'est au joueur 2 de jouer. S'il s'avère que le coup proposé par le joueur est invalide, la partie s'arrête.

A la fin de la partie, le serveur libère toutes les variables allouées dynamiquement ainsi que les bibliothèques compilées des joueurs.

2.2.2 Joueurs

Au cours de ce projet, cinq joueurs différents ont été élaborés, possédant chacun leur propre stratégie.

Les cinq joueurs sont en réalité des bibliothèques que le serveur utilise pour les besoins du jeu (cf section 2.2.1). En revanche, chaque joueur a accès à une copie du graphe qu'il gère lui-même afin de savoir quelles sont les cases libres ou occupées.

Le **premier joueur** implémenté s'intitule `z_random_player`[1] car il s'agit d'un joueur à la **stratégie aléatoire**. Ainsi, ce joueur propose un sommet au hasard parmi les sommets libres à chaque coup, que ce soit pour initier le jeu ou le poursuivre. De la même façon, il accepte ou refuse aléatoirement le coup initial de son adversaire.

Ce joueur a posé un problème intéressant. En effet, sa première implémentation le faisait tirer de façon naïve un nombre aléatoire compris entre 0 et le nombre de sommets, puis une vérification de la validité du sommet était réalisée. Cette technique a vite trouvé ses limites sur des grands plateaux et lorsqu'il ne restait plus que quelques cases. En effet, le joueur, adoptant un comportement de tirage aléatoire avec remise, pouvait bloquer le jeu longtemps avant de trouver un numéro de sommet inoccupé, provoquant de ce fait un *time out*. Pour gérer ce problème, une liste des sommets non occupés a été dressée, simulant ainsi un tirage aléatoire sans remise et évitant le *time out*.

La complexité en temps associée à ce joueur est **linéaire** $\mathcal{O}(n)$, puisqu'il n'existe qu'un parcours de liste pour déterminer le sommet à jouer.

Le **second joueur** s'intitule `min_dist_player`. L'objectif de celui-ci est de **minimiser la distance** existante entre les deux composantes connexes de départ. Ainsi, le joueur va réduire petit à petit la distance entre les deux bords du plateau en avançant de préférence en ligne droite. Concrètement, cette stratégie consiste à choisir un sommet adjacent par rapport au dernier joué. Pour ce faire, on utilise un tableau qui va recenser tous les voisins du dernier coup joué, et ainsi déterminer la case qui va rapprocher le plus possible les deux composantes connexes principales. Si aucune case n'est adéquate, alors le joueur joue aléatoirement. La complexité en temps associée à ce joueur pour choisir un coup avoisine :

$$\mathcal{O}(n^{3/2} + 4\sqrt{n} * m + comp(Random\_player)) \tag{3}$$

avec m la taille du tableau qui contient les voisins, et n le nombre de sommets. En effet, le tableau est parcouru 4 fois dans le pire des cas pour vérifier la possibilité de jouer certains coups, et celui-ci peut être réaliser $\sqrt{n}$ fois si aucun coup n'est valide. De plus, si la stratégie n'est pas validée, alors le joueur joue aléatoirement. La complexité en temps de ce joueur est donc **inférieure à une complexité quadratique**.

Notre **troisième joueur** est une amélioration directe de `min_dist_player`. Il se nomme `virtual_connections_player`. Il utilise une stratégie adaptative qui ne

1. Le préfixe `z_` est un choix dû au comportement de la Forge. Celle-ci ne faisant jouer que les trois premiers joueur par ordre alphabétique, nous avons de cette façon "invisibilisé" `random_player` vers la fin de notre projet.

fonctionne que sur les plateaux hexagonaux : celle des **connexions virtuelles**. En effet, sur ces plateaux, pour se déplacer vers une direction cardinale à partir d'un sommet n, le joueur à toujours deux choix qui s'avèrent équivalents (cf figure 2 où le joueur noir se déplace vers le bas : il peut suivre de façon équivalente la flèche de gauche ou la flèche de droite). Puis, au coup suivant, il est obligé de prendre la case dans l'alignement de la case n s'il veut avancer rapidement. La technique des connexions virtuelles consiste à progresser en évitant de choisir les "carrefours" (Figure 2). De cette façon, le joueur s'assure un chemin virtuel. Lorsque la ligne droite discontinue est formée, il ne reste plus qu'à combler les vides au niveau des "carrefours" (figure 3). Cependant, il est nécessaire de ne pas combler ces connexions virtuelles uniquement lorsque le chemin principal est formé. En effet, durant la partie, si le joueur adverse bloque l'un des deux chemins possibles, il faut alors choisir le chemin libre au tour suivant sous peine de ne plus pouvoir relier les sommets.

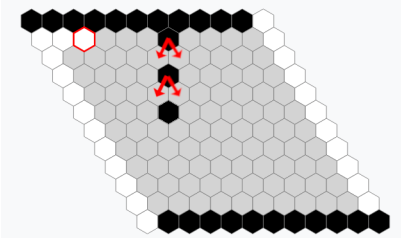

FIGURE 2 – Étape 1 : progression discontinue, sauvegarde des "carrefours"

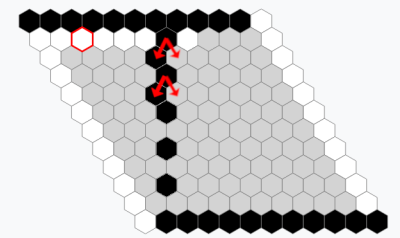

FIGURE 3 – Étape 2 : complétion du parcours via les "carrefours"

Au niveau de notre implémentation, nous utilisons une matrice en variable globale qui va sauvegarder les sommets des "carrefours". Cela nous permet de vérifier à chaque tour si l'adversaire a joué sur un des "carrefours", le mettant alors en danger.

Si jamais les conditions de la méthode des connexions virtuelles ne sont pas validées (par exemple sur un plateau carré ou triangulaire), alors on utilise la technique de la distance minimale. La complexité en temps associée à ce joueur pour choisir un coup est du même ordre que pour le joueur `min_dist_player`.

Le **quatrième joueur** est une amélioration de `virtual_connections_player` et se nomme `tsunami`. Cette heuristique tente de trouver un autre passage utilisant les connexions virtuelles si celui qu'elle suivait initialement se retrouve bloqué. En effet, tout sommet possédant six voisins admet six possibilités de connexions virtuelles. Ainsi, nous choisissons la connexion disponible permettant de réduire la distance entre les deux bords du graphe. Par conséquent, cet outil s'avère plus efficace que la stratégie du troisième joueur puisqu'il permet de gagner une case de plus.

Au niveau de notre implémentation, nous avons défini une structure qui contient la liste des sommets des "carrefours" ainsi que sa taille. Cela nous permet d'améliorer l'ancienne implémentation (utilisation de matrice) et faciliter l'ajout et la suppression des sommets.

Si jamais aucun choix n'est possible, alors nous utilisons la technique de la distance minimale.

La complexité en temps associée à ce joueur est :

$$\mathcal{O}(n^{3/2} + 12 * l + 4 * n^{1/2} * m + comp(Random\_player)) \tag{4}$$

avec l la taille de l'ensemble relatif aux connexions virtuelles, m la taille du tableau qui contient les voisins, et n le nombre de sommets. En effet, il y a quatre possibilités à étudier à chaque itération, et chacune des ces possibilités parcours le tableau des connexions virtuelles trois fois. La complexité en temps de ce joueur est donc inférieure à une complexité quadratique.

Le **dernier joueur** que nous avons développé utilise des calculs de résistance inspirés des travaux de **Gustav Kirchhoff** et de l'article scientifique de Vadim V. Anshelevich : *The Game of Hex : An Automatic Theorem Proving Approach to GameProgramming*. Nous ne sommes pas parvenu à l'implémenter entièrement au cours de ce projet.

Le principe est de considérer le plateau comme un circuit électrique dont les connexions entre deux sommets forment des résistances. Ainsi, on distingue trois type de résistances :

- **les résistantes nulles** qui appartiennent au joueur. La case appartient déjà au joueur et y passer ne coûte plus rien (ces résistances ne sont pas réellement nulles dans notre projet, mais limitées à une valeur `epsilon` $= 0.01$)
- **les résistances infinies** qui appartiennent à l'adversaire. Y passer est impossible. Nous modélisons l'infini par une valeur de résistance de 100.
- **les résistances classiques** dont la valeur est égale pour le joueur comme pour l'adversaire. Il est possible d'y passer, mais cela coûtera un tour. La valeur de ces résistances est de 1.

Ainsi, le but du jeu est de faire tendre le ratio $E = \frac{R_B}{R_W}$ vers 0 pour le joueur noir et vers $+\infty$ pour le joueur blanc, où R_B est la résistance du joueur noir et R_W est la résistance du joueur blanc.

Afin de déterminer la résistance globale d'un circuit, nous avons utilisé la loi des mailles, celle-ci stipulant que dans une maille d'un réseau électrique, la somme des tensions est toujours nulle. La figure 4 ci-dessous permet d'illustrer le problème.

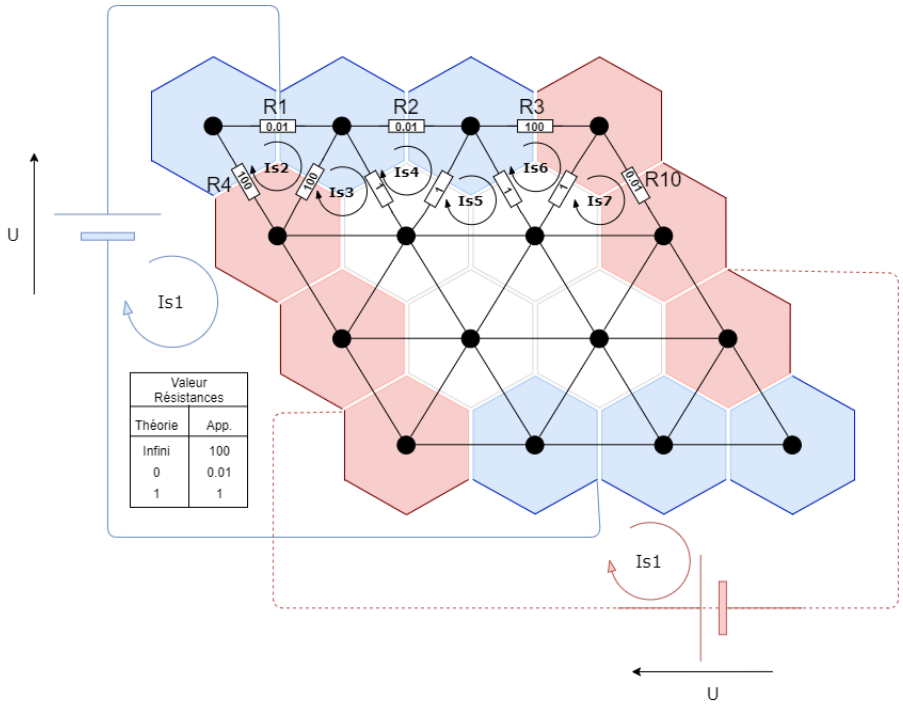

FIGURE 4 – Représentation du circuit électrique sur un plateau de jeu.

Le but est alors de résoudre le système 5 ci-dessous (cas du circuit électrique bleu sur la Figure 4) :

$$\begin{cases} Mesh1: & R_1(I_{s1}-I_{s2})+R_4(I_{s1}-I_{s2})+R_{14}(I_{s1}-I_{s8}) \\ & +R_{24}(I_{s1}-I_{s14})+R_{31}(I_{s1}-I_{s15})=0 \\ Mesh2: & R_1(I_{s2}-I_{s1})+R_4(I_{s2}-I_{s1})+R_5(I_{s2}-I_{s3})=0 \\ Mesh3: & R_5(I_{s3}-I_{s2})+R_6(I_{s3}-I_{s4})+R_{11}(I_{s3}-I_{s8})=0 \\ ... & \end{cases} \tag{5}$$

Afin de résoudre ce système, il est utile de générer l'ensemble des cycles minimaux du graphe. Cependant, déterminer cet ensemble n'était pas envisageable en temps raisonnable à notre niveau. Nous avons donc opté pour la recherche d'une **base de cycles**. Cette méthode ne garantit pas le résultat optimal que proposerait l'ensemble des cycles minimaux, cependant celle-ci possède un nombre d'équations suffisant pour résoudre le système linéaire.

Pour réaliser cette génération, un **arbre couvrant** est utilisé. Cet arbre est stocké dans une matrice d'adjacence. Par définition un arbre couvrant ne contient pas de cycle, ainsi tout ajout d'arête en entraînerait la formation. La méthode que nous avons choisie pour déterminer les cycles se base sur ce principe.

Dans les faits, on génère notre arbre couvrant et on le compare à la matrice d'adjacence du plateau. L'objectif est de récupérer toutes les arêtes faisant partie de la matrice et n'étant pas dans l'arbre. Pour cela, on fait simplement une différence matricielle entre la matrice du plateau et la matrice de l'arbre ce qui permet d'obtenir le résultat voulu. Enfin, chaque cycle découvert est enregistré dans une structure prévue à cet effet.

Pour être en mesure de résoudre l'ensemble des équations du système, il a également fallu réaliser une structure permettant de sauvegarder chaque arête et la liste des cycles auxquelles elles appartenaient.

Grâce à l'ensemble des éléments précédemment évoqués, il est possible de **générer les équations** dans une matrice de la bibliothèque `GSL` [2]. Le but est alors d'utiliser le solveur de la bibliothèque pour résoudre le système en temps cubique $\Theta(n^3)$. Lorsque la solution pour I_{s1} est déterminée, il suffit d'appliquer la formule $R = \frac{U}{I}$ pour trouver la résistance induite par le joueur. Il faut alors réaliser le même procédé pour le circuit du joueur adverse et ainsi déterminer la valeur du ratio E évoqué précédemment. Enfin, pour déterminer le meilleur coup, il faut réaliser ces opérations autant de fois qu'il y a de coups possibles à jouer et donc retrouver celui qui propose la meilleure solution.

2. NB : Cette description correspond aux intentions d'implémentation non réalisées faute de temps.

2.3 Tests et documentation

Afin de réaliser nos tests, nous avons créé un fichier test pour chacun de nos fichiers sources dans la mesure du possible. Dans ces fichiers, nous réalisons des tests sur chacune de nos fonctions. Nous compilons tous ces fichiers tests en fichiers objets que nous lions dans un unique exécutable `alltests` qui permet l'exécution de tous nos tests. Au cours de nos tests, nous utilisons notamment la bibliothèque *assert.h*.

Nous couplons nos tests unitaires avec une analyse *valgrind* lors de leur exécution.

En ce qui concerne la documentation, nous avons commenté les fonctions dans notre code de façon à pouvoir exécuter un fichier *Doxyfile*. Ainsi, il est possible d'obtenir une documentation *Doxygen* de tout notre projet.

2.4 Amélioration

Actuellement, certaines améliorations peuvent être apportées au code. D'une part, au delà de la terminaison de la stratégie de jeu utilisant l'heuristique de Kirchhoff, certaines de nos fonctions méritent une revue afin de baisser leur complexité. En effet, si l'on s'attarde sur l'annexe 8, nous pouvons remarquer que l'appel à la fonction `create_hexagonal_graph` est coûteux en nombre d'exécutions. La figure 5 illustre bien le problème.

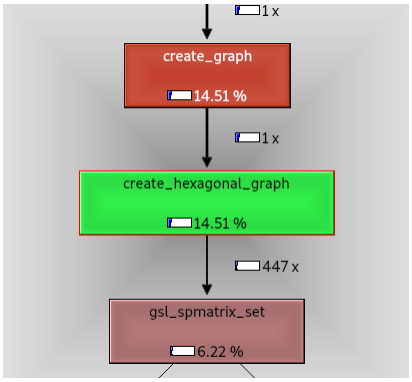

FIGURE 5 – Coût de l'appel de la fonction `create_hexagonal_graph() pour une taille de graphe` $m = 8$ (KCaChegrind)

Ainsi, on remarque nettement que l'affectation d'une valeur dans la matrice est répétée 447 fois. Ceci est due à un problème d'implémentation. En effet, l'affectation des valeurs pour cette matrice a été pensée par le biais d'une imbrication de deux

boucles itératives donc nous avons une complexité en temps quadratique $\mathcal{O}(n^2)$, alors qu'une unique boucle aurait été suffisante, permettant ainsi une complexité en temps linéaire.

Une autre amélioration possible concerne notre couverture de tests. En effet celle-ci est très satisfaisante pour la quasi totalité de nos fichiers (couverture $\geq 90\%$), cependant l'un de ces fichiers ne possède qu'une couverture de 30% ce qui s'avère trop faible pour assurer son bon fonctionnement. Enfin, il s'avère que certains fichiers ont encore une **complexité cyclomatique** un peu trop élevée, ce qui dénote qu'il est encore possible de rendre certaines fonctions plus atomiques.

3 Conclusion

Finalement, au cours de ce projet nous avons implémenté un serveur fonctionnel permettant de faire s'affronter deux bibliothèques de joueurs. Nous avons codé nos propres joueurs obéissant à différentes stratégies en partant de la plus simple pour tendre vers des techniques plus élaborées.

Nous avons conscience des limites et des atouts de notre implémentation. Nous avons veillé à rendre notre code le plus générique possible et à soigner sa complexité. Cependant, il existe des points d'améliorations comme la possibilité d'améliorer les complexités des créations des graphes ainsi que la complexité cyclomatique.

Ce projet a été très formateur. D'une part, il nous a permis d'améliorer notre maîtrise du langage et de ses outils et d'autre part, il nous a permis d'apprendre à travailler en équipe.

Annexes

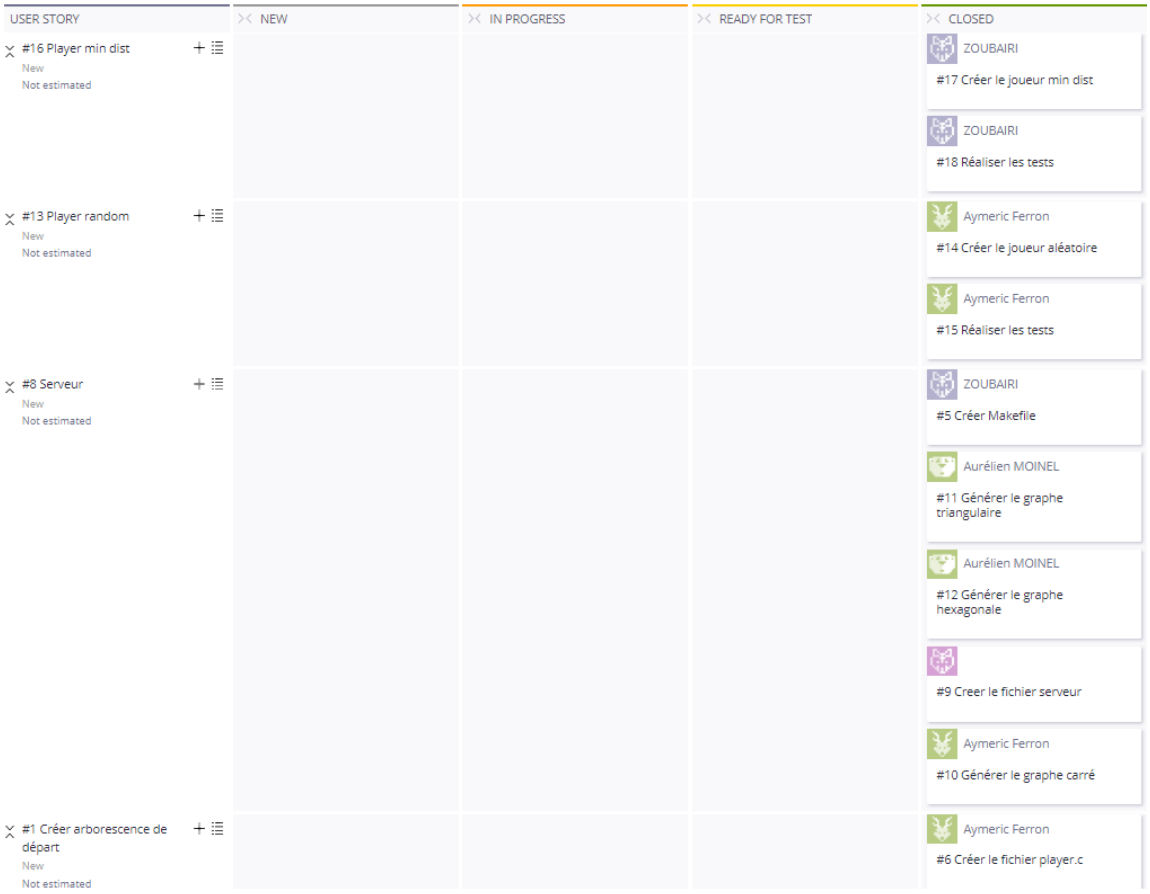

FIGURE 6 – Interface de gestion de projet Taïga

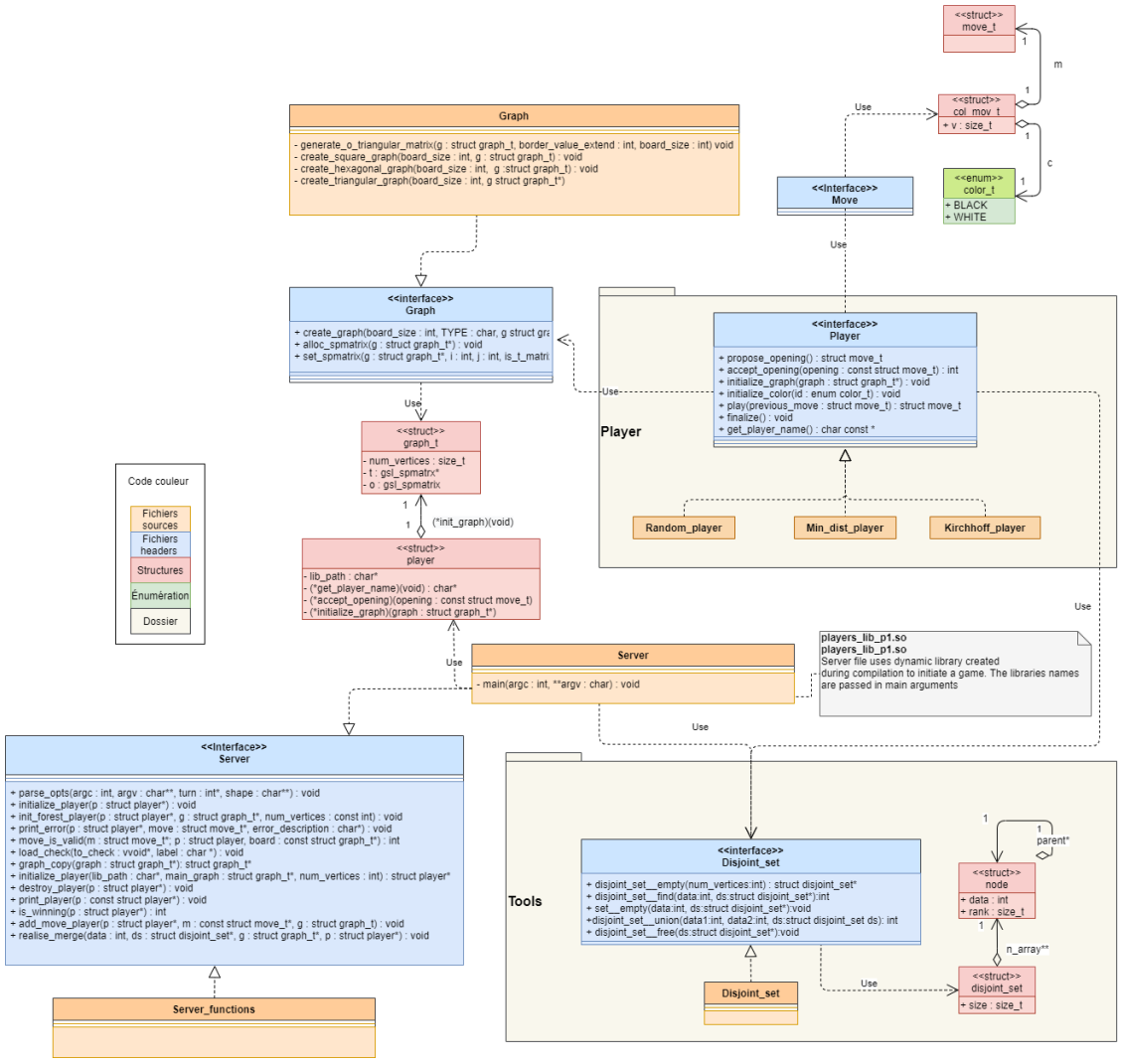

FIGURE 7 – Diagramme de classes jeux de Hex

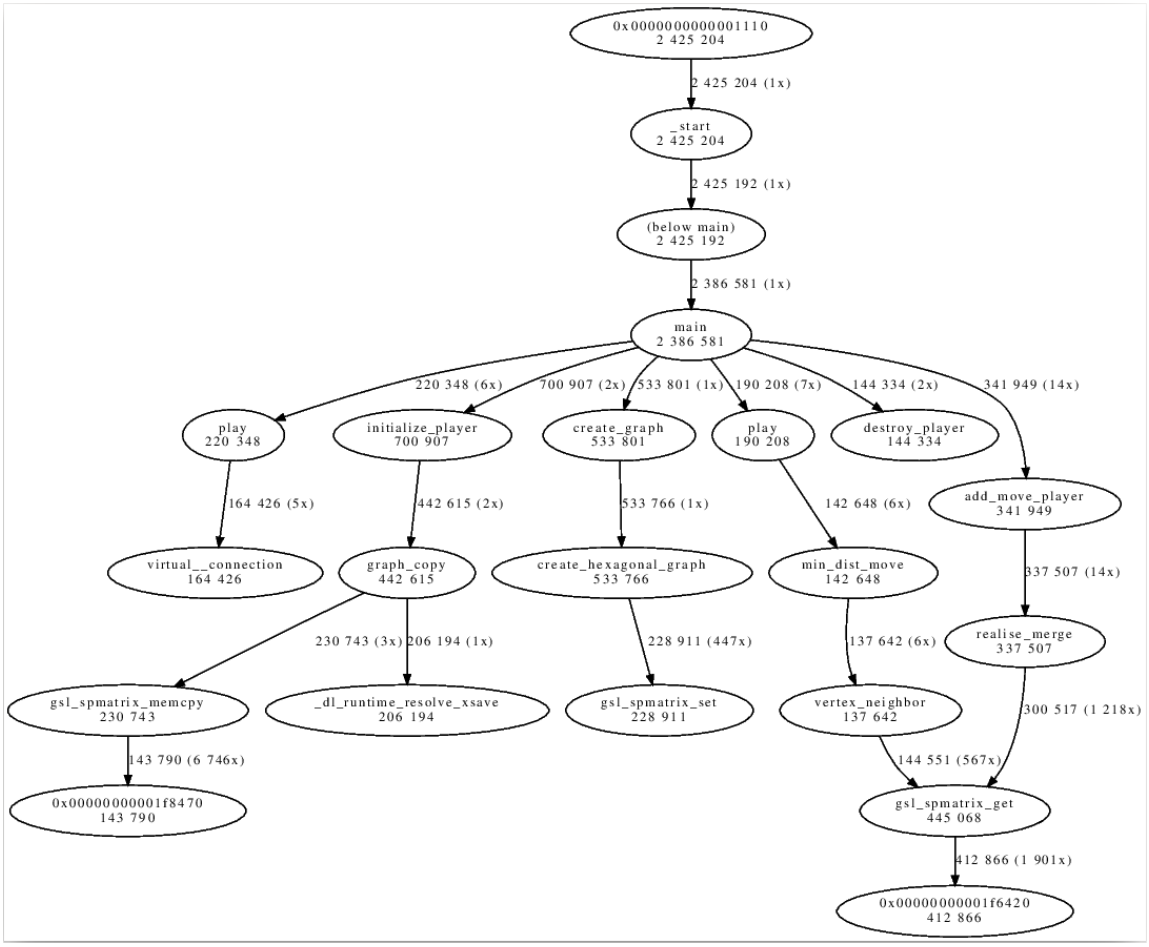

FIGURE 8 – Arbre des appels lors du lancement d'une partie entre deux joueurs (KCacheGrind)

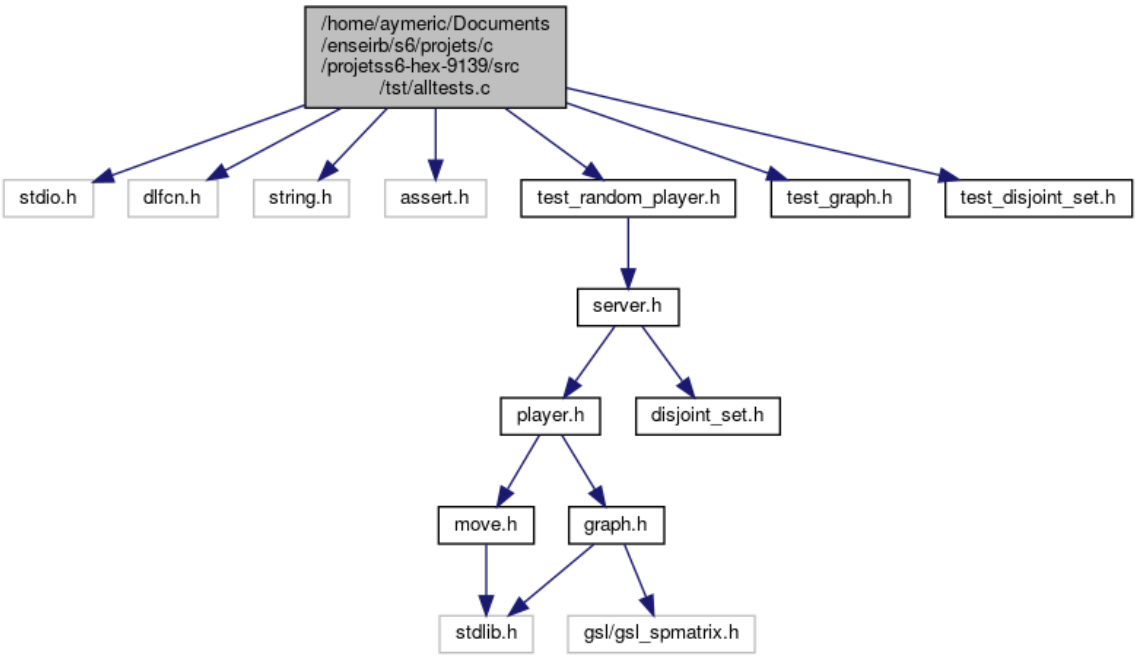

FIGURE 9 – Diagramme d'inclusion du fichier alltests.c

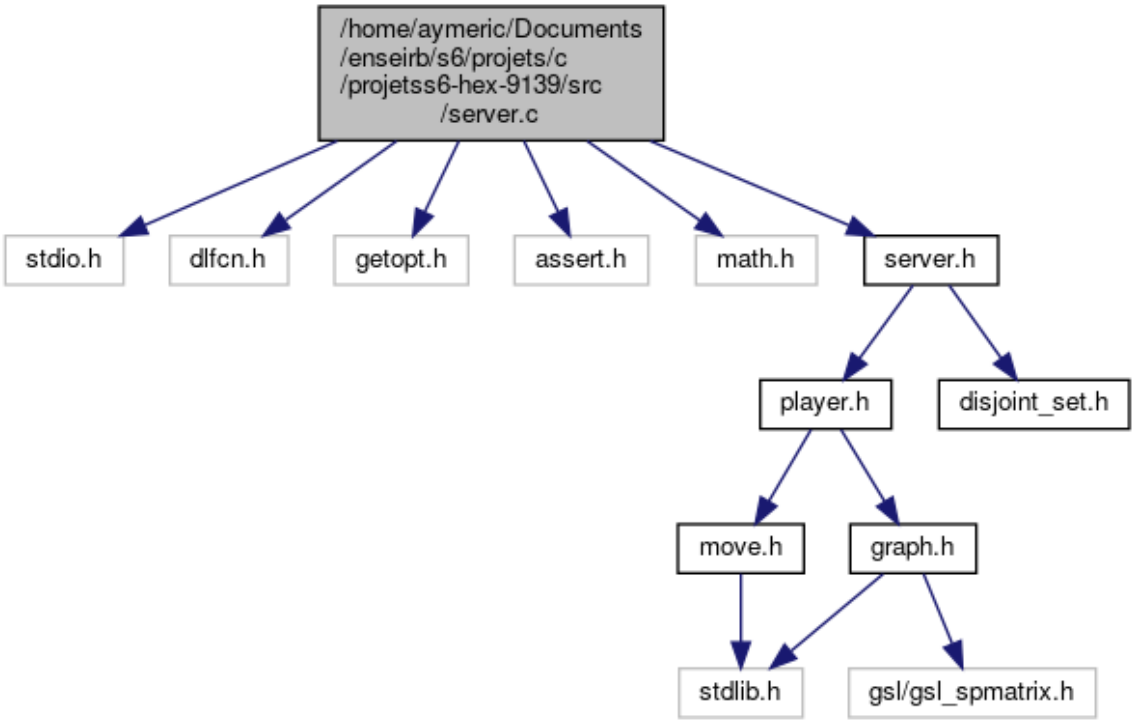

FIGURE 10 – Diagramme d'inclusion du fichier server.c