HEX

Projet réalisé par

Boullit Mohamed Fayçal
Izekki Jalal
Moussa Guimba Mamadou
Pierre Hugo

Projet encadré par

Herbreteau Frédéric

Equipe 9151

Année 2019/2020

Table des matières

1 Introduction

Dans le cadre du second projet de première année, filière informatique, il nous est proposé de mettre en application nos compétences en programmation en implémentant le jeu Hex, écrit en langage C.
Hex est un jeu de société opposant deux joueurs, qui se joue sur un tablier formé par des hexagones réguliers. Les joueurs sont représentés chacun par une couleur et possèdent des pions de leur couleur qu'ils disposent un à un sur les cases vides de leur choix du plateau. L'objectif de tout joueur est de réussir à construire un chemin continue de pions reliant les deux bords de sa couleur.
Nous allons vous faire découvrir en profondeur le jeu de Hex dans ce projet et les différentes stratégies de jeu permettant de remporter la partie. Pour ce faire nous allons mettre en place un ensemble de clients possédant une interface commune mais qui ont leur propre plateau de jeu et jouant en fonction de leur propre stratégie et un serveur qui permet l'interaction entre deux clients.

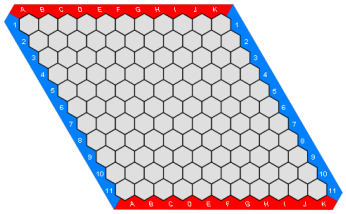

FIGURE 1 – Tablier du jeu vide

2 Le plateau de Jeu

2.1 Représentation du plateau

Le plateau de jeu est représenté par un graphe dont les noeuds sont exactement les cases du palier. Pour manipuler le graphe, la bibliothéque GSL (GNU Scientific Library) est utilisée. C'est une bibliothéque numérique pour les programmeurs C et C ++ qui fournit dans le cadre de ce projet des fonctions qui manipulent des matrices creuses. Ce qui permet de modéliser le graphe en considérant une matrice d'adjacence t de taille $n \times n$ qui vérifie la propriété t[i][j] = 1 si et seulement s'il existe une arrête entre le noeud i et le noeud j où n est le nombre de sommet du graphe. Une autre matrice o est définie de taille $2 \times n$ qui vérifie que o[k][i] = 1 si et seulement si le noeud i est occupé par le joueur k. Plus précisément, une structure nommé **graph_t** représente le graphe. Elle est constituée du nombre total de noeud et des deux matrices creuses citées précédemment :

```
    struct graph_t {
  size_t num_vertices;   // Number of vertices in the
  gsl_spmatrix* t;
  gsl_spmatrix* o;
};
```

Dans ce projet, nous travaillons avec trois types de graphe de taille variable à savoir le graphe hexagonal, carré et triangulaire représentés sur la figure suivante.

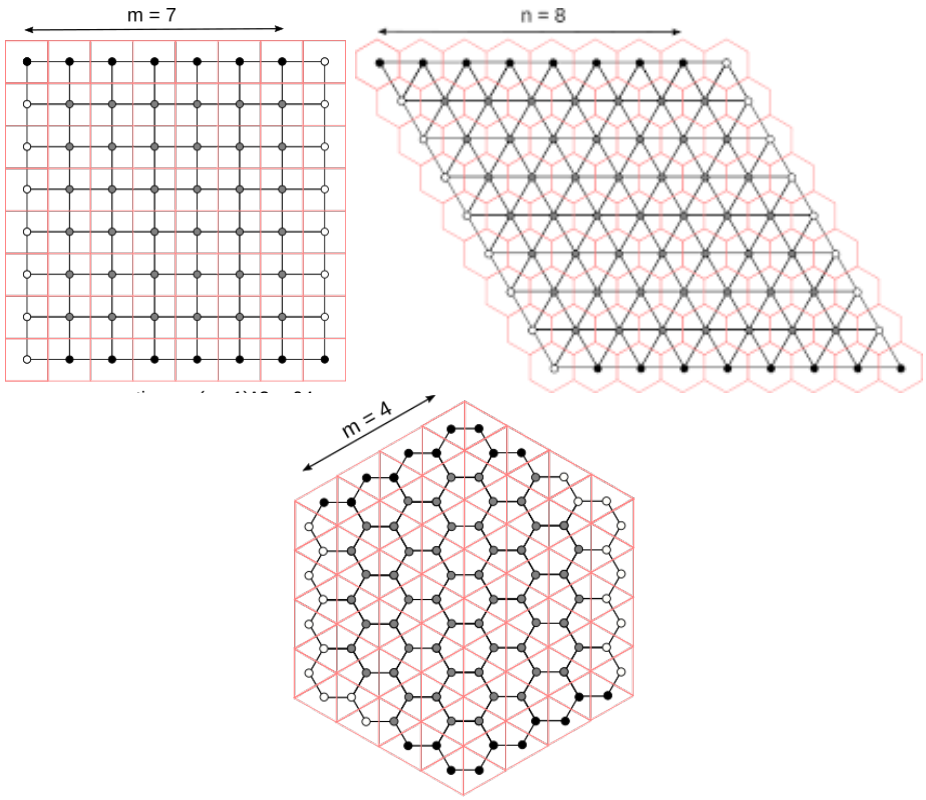

FIGURE 2 – Les types de graphe utilisés

2.2 Les coups de jeu

Nous représentons un coup de jeu par une structure **struct move_t** qui contient le numéro du noeud choisi par le joueur ainsi que la couleur du joueur.

```
struct move_t {
size_t m;
enum color_t c;
```

On choisi les couleurs : BLACK = 0 et WHITE = 1 pour désigner les joueurs. Le joueur de couleur Noir est celui qui commence la partie de jeu.

2.3 Les coups de jeu

3 Mise en oeuvre technique

3.1 Les joueurs

Les différents joueurs sont codés indépendamment et sont sauvegardés dans des bibliothèques dynamiques qui pourront être chargées dynamiquement par le serveur. Afin de définir l'inter-fonctionnalité entre les différents clients, une même interface est requise pour tous les joueurs.
Tout joueur doit sauvegarder le graphe que lui passe le serveur et doit être capable de proposer et d'accepter ou non un coup en fonction de la situation dans laquelle il se trouve. C'est ce que font respectivement les fonctions **initialize_graph**, **propose_opening** et **accept_opening**. Vient alors la fonction **initialize_color** qui assigne à chaque joueur sa couleur et la fonction **play** qui permet au joueur de proposer son coup en fonction de sa stratégie.

3.2 Le serveur

Le serveur est le programme qui permet le déroulé du jeu, en envoyant les informations aux stratégies et en vérifiant l'avancement de la partie.

3.2.1 La boucle de jeu

La boucle de jeu est toute simple, dans un premier temps une copie du graphe représentant le plateau de jeu est envoyé à chacun des joueurs, ce ne sont que des copies afin d'éviter toute triche qui consisterait à modifier le plateau de jeu en changeant par exemple des cases déjà colorées.

Le démarrage de la partie se fait selon la règle dit du gâteau, elle consiste à demander à un joueur de jouer un premier coup, puis de demander au second joueur si il veut prendre ce coup pour lui ou le laisser à l'autre joueur, le serveur attribue ainsi leur couleur à chaque joueur, le joueur ayant le premier coup est le joueur noir.

Vient ensuite la boucle de jeu à proprement parler, cette dernière consiste en une boucle tant que vérifiant que le graphe n'est pas entièrement coloré. À chaque tour le serveur calcule le prochain joueur puis lui fait jouer son coup et enfin fait appel à la fonction **is_winning** vérifie si la partie est terminée. Cette fonction renvoie un booléen indiquant si la partie est terminé ce qui de stopper la boucle à l'aide d'un **break**.

```
1  G ← graph__initialize(M, T);
2  for tout joueur p do
3  |  p->initialize__graph(G)
4  move = joeur1->propose_opening();
5  if joeur2->accept_ opening(move) then
6  |  C'est le joeur2 qui joue le prochain coup donc
7  |  joueur1->initialize_color(BLACK);
8  |  joueur2->initialize_color(WHITE);
9  else
10 |  C'est le joeur1 qui joue encore une fois et donc
11 |  joueur1->initialize_color(WHITE)
12 |  joueur2->initialize_color(BLACK);
13 while G n'est pas rempli do
14 |  p ← le joueur suivant;
15 |  move = p->play(move);
16 |  if Il gagne avec le noeud move then
17 |  |  break
18 |  else
   |  _
```

Algorithme 1 : Principe algorithmique de la boucle de jeu

3.2.2 La fonction is_winning

Cette fonction est celle permettant au serveur de savoir si la partie est terminée, mais elle est également celle qui colore le graphe pour le serveur. Pour ce faire, elle prend en paramètre un pointeur vers une structure **struct game** qui réunie le plateau de jeu ainsi que les joueurs représentés par la structure **struct player** (voir section 3.2.3), le coup jouer représenté par la structure **struct move**, ainsi

qu'un caractère indiquant la forme du plateau de jeu (voir section 2.1) et la taille du graphe.

Dans un premier temps, la fonction vérifie si le coup est possible et si oui colore le graphe en fonction.

Vient alors la vérification pour savoir si le dernier joueur ayant jouer a gagné la partie. Cela se fait par un parcours en profondeur sur le sous-graphe de la couleur de ce joueur afin de repérer une éventuelle connexion entre les deux partie colorées d'origines du joueur. Pour ce faire, il y a deux fonctions qui calculent les indices d'un nœuds dans une partie et dans l'autre, pour cela elles prennent en paramètre la forme du graphe, sa taille ainsi que la couleur testée. Le premier nœud sert de départ pour le parcours en profondeur et si le deuxième nœud est rencontré lors de ce parcours alors les deux parties sont reliées et le joueur à gagner.

3.2.3 Communication avec les joueurs

Comme vu dans la section 3.1, les joueurs sont codés dans des bibliothèques appelées dynamiquement. Cet appel est réalisé grâce à la bibliothèque **dlfcn** qui permet d'ouvrir dynamiquement une bibliothèque. Les stratégies sont donc données en paramètre lors du lancement du serveur.

Une structure **struct player** réunie les pointeurs des fonctions fournies par la bibliothèque du joueur (partie **??**) ainsi que le pointeur vers cette bibliothèque pour pouvoir la fermer à la fin de la partie.

3.3 Implémentation des graphes

La structure **graph_t** évoquée dans la partie 2.1, nécessite plusieurs fonctions pour la manipuler, et parmi les fonctions principales on trouve **graph_initialize** une fonction qui prend en paramètre la longueur du tablier souhaité, et la forme (hexagonal,carré ou triangulaire) et qui retourne un pointeur vers une structure **graph_t** allouée et bien initialisée. La seconde fonction qui réside importante c'est la fonction qui permet au serveur de créer des copies pour les joueurs **graph_copy**, ainsi qu'une liste de fonction qui ne laisse plus le besoin de faire appel à une fonction de la bibliothèque GSL en dehors du fichier *graph_function.c*. Le tablier est affiché par la fonction **graph_print** qui prend le graphe en paramètre, et sa forme pour l'afficher correctement.

4 Les différentes stratégies de jeu

4.1 La stratégie aléatoire

La stratégie aléatoire consiste à proposer un coup objectivement parmi les coups possibles. S'il arrive qu'il débute la partie, le coup qu'il propose est également aléatoire sinon, il accepte toujours le coup joué par l'adversaire. Ainsi, le principe algorithmique sur lequel repose la stratégie aléatoire est le suivant :

Data : Le dernier coup **pm** du joueur adverse
Result : Le coup **nm** du joeur aléatoire
1 marquer dans le graphe le coup du joueur adverse ;
2 choisir aléatoirement un noeud **nm** quelconque du plateau jeu
3 **while** *le noeud* ***nm*** *est occupé* **do**
4 choisir aléatoirement un autre noeud **nm**
5 colorer dans le graphe le noeud choisi ;
6 retourner le noeud

Algorithme 2 : Principe algorithmique du coup aléatoire

4.2 La stratégie du bloqueur

Le joueur qui utilise cette stratégie de jeu a pour but de bloquer les coups de son adversaire. En effet, comme l'adversaire essaye de joindre les deux bords du plateau correspondant à sa couleur par un chemin continue, l'objectif du bloqueur est de mettre un obstacle devant chacun de ses coups joués. Cependant le choix des noeuds à occuper dépend de la couleur avec la quelle joue le client stratège. Considérons par exemple un graphe hexagonale comme schématisé sur la figure 3 :

- Si le bleu représente l'adversaire, bloquer son coup revient à choisir l'un des noeuds blancs(C'est celui de gauche qui est généralement choisit). S'il arrive qu'ils soient occupés, le joueur rouge choisit un parmi les noeuds marrons mais généralement c'est le premier à gauche qui est choisit.
- Par contre si le rouge représente l'adversaire, la stratégie est de privilégier les noeuds marrons aux noeuds blancs.

S'il s'avère que tous les voisins du coup joué par l'adversaire sont colorés, le joueur propose aléatoirement un coup parmi ceux possibles.

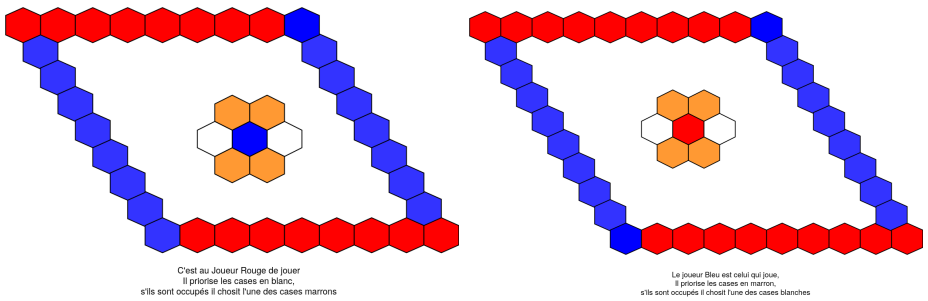

FIGURE 3 – Choix du noeud obstacle

Ainsi, le principe algorithmique sur lequel repose cette stratégie est le suivant :

```
   Data : Le dernier coup pm du joueur adverse
   Result : Le coup nm du joeur aléatoire
 1 marquer dans le graphe le coup pm joué par l'adversaire ;
 2 Soit T un ensemble vide ;
 3 T ← tous les noeuds inoccupés qui sont voisins à pm ;
 4 if T n'est pas vide then
 5 |   if Le joueur qui joue est le rouge then
 6 |   |   privilégier le noeud de gauche et celui de droite du noeud rouge ;
 7 |   |   s'ils sont colorés choisir un noeud non coloré parmi les quatre autres ;
 8 |   else
 9 |   |   privilégier les noeuds placés au dessus et en dessous du noeud bleu ;
10 |   |   s'ils sont tous occupés, choisir soit celui de gauche ou celui de droite ;
11 else
12 |   choisir aléatoirement un noeud inoccupée
13 colorer dans le graphe le noeud choisi ;
14 retourner le noeud
```

Algorithme 3 : Principe algorithmique de la stratégie du bloqueur

4.3 la stratégie du coup maintenant le plus court chemin

Cette stratégie se base essentiellement sur le maintien du plus court chemin après chaque coup joué par l'adversaire. En effet une structure stratégie est nécessaire pour stocker les indices des bords du plateau au début du jeu qui permettront de déterminer à chaque fois le chemin le plus court à travers le fameux algorithme *DIJSKTRA*. A chaque fois qu'un chemin est généré on suppose que n est la distance ou bien le nombre de cases restantes pour ce chemin, à ce stade pour trouver le prochain coup, on colore chaque case du chemin par la couleur de l'adversaire si jamais on ne trouve pas un chemin alternative de profondeur n, donc cette case colorée est la seule à avoir maintenu la notion du plus court chemin, c'est bien la case à jouer.

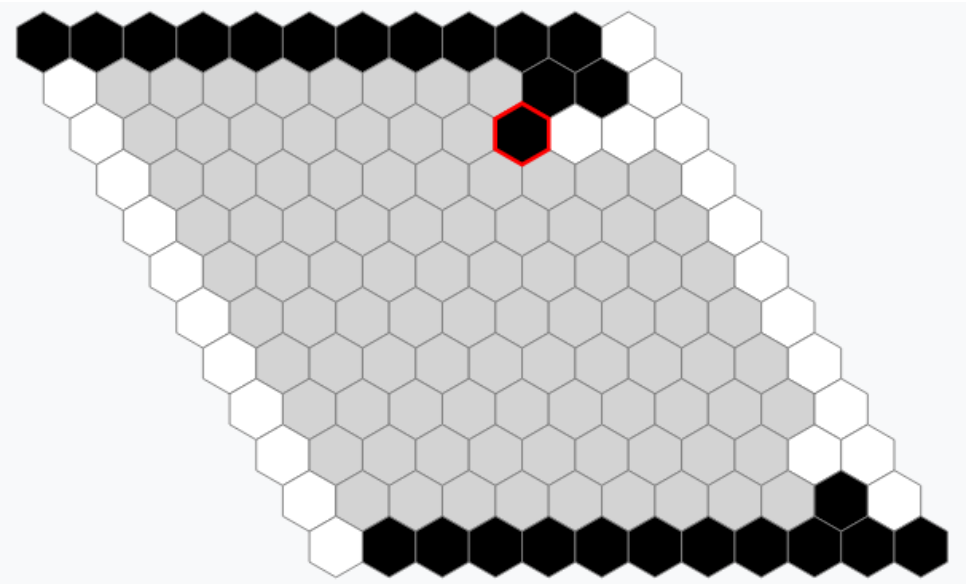

FIGURE 4 – Prochain coup généré par l'algorithme pour le joueur noir

La figure 4.3 montre le coup choisi par l'algorithme, avant de jouer le coup choisi

entouré en rouge, le plus court chemin était de profondeur 9, et ce coup étant un parmi les coups minimisants la profondeur du plus court chemin, mais étant le seul à avoir garder plus de 2 chemins de profondeur 8 par suite, en supposant n'importe quel prochain coup de l'adversaire dans le cas de la figure 4.3 le joueur garde toujours un second chemin de profondeur 8 à suivre.

```
   Data : Tablier du jeu T, Couleur du joueur c
   Result : Le coup n
 1 Chemin ← plus court chemin entre les deux bords colorés en c de T ;
 2 profondeur ← taille(chemin) ;
 3 if Chemin est vide then
 4 |  n ← coup aléatoire valide ;
 5 while Chemin !vide do
 6 |  n ← noeud du Chemin ;
 7 |  Retirer n du Chemin ;
 8 |  Colorer le noeud n par la couleur inverse de c ;
 9 |  alternative ← plus court chemin entre les deux bords ;
10 |  if profondeur < taille(alternative) then
11 |  |  break
12 colorer dans le graphe n choisi par la couleur c ;
13 retourner n ;
```

Algorithme 4 : Principe algorithmique du coup maintenant le plus court chemin

La complexité de l'algorithme est **quadratique** en nombre de sommets, dû à celle de *Dijsktra* implémenté en utilisant des tableaux. Ce qui demande un temps raisonnable au joueur pour répondre au coup de l'adversaire.

4.4 la stratégie "two-distances"

La stratégie implémenté basée sur le principe du deux-distances *(two-distances, van Rijswijck, 2000)* est fournit principalement par l'algorithme *Minimax*. En effet,*two-distances* est une façon d'évaluer un tablier. En l'évaluant, on peut déterminer le joueur qui est susceptible de gagner. Le problème avec la stratégie précédente c'est que si l'adversaire bloque le joueur un peu plus loin, le joueur n'a aucune information sur l'existence de son plus court chemin et du chemin alternatif, d'où l'idée d'évaluer plus de positions.

4.4.1 Méthode de Minimax

La stratégie minimax détermine les valeurs des positions du tablier non terminales ou détermine de nouvelles positions terminaux. L'algorithme suppose un jeu optimal alternatif des deux côtés et recherche les branches de l'arbre vers les nœuds terminaux, ou jusqu'à une profondeur donnée. Dans le cas du jeu de Hex la taille du tablier ne permet pas d'aller jusqu'au bords, d'où la nécessité d'une fonction qui évalue un tablier avec quelques coups.

4.4.2 Évaluation "two-distances"

La fonction d'évaluation *two-distances* est basée sur la mesure de la connectivité relative d'une position Hex, et qui prend en considération la présence des chemin

alternatifs. Son principe définit la distance $d(u, v)$ pour deux sommets u et v comme suit :

$$d(u, v) = 1 \ , \ Si \ u \ et \ v \ sont \ voisins. \tag{1}$$

Sinon, en supposant $N(u)$ l'ensemble des voisins de u, et en retirant u' le voisin de u de plus proche distance de v, on considère un nouveau voisin $u'' \in N(u)$ le plus proche de v, et on définit :

$$d(u, v) = 1 + d(u'', v) \tag{2}$$

Alors, en considérant e_{1p} e_{2p} deux sommets appartenant respectivement aux deux bords du tablier de la même couleur pour un joueur p, la fonction d'évaluation du tableau f est la distance entre les deux sommets :

$$f(p) = d(e_{1p}, e_{2p}) \tag{3}$$

4.4.3 Principe implémenté

Le principe implémenté et décrit par l'algorithme 5 consiste à minimiser la distance entre les deux bords du tablier du joueur en jouant un coup. Par l'algorithme de *minimax*, on souhaite évaluer les noeuds du tablier, l'évaluation d'un noeud dans un tablier vide de forme hexagonale coûte $\mathcal{O}(6^d \times c)$, où c est la complexité de la fonction d'évaluation, d la profondeur souhaitée lors de l'utilisation de *mimimax*. En effet, une grande complexité résulte de la fonction d'évaluation où on utilise *DIJSKTRA* pour trouver les voisins u' et u" ce qui rend l'algorithme plus complexe.
Alors, afin de minimiser le temps de calcul du joueur, un choix optimal des noeuds évalués est fait, il consiste à ne prendre en considération que l'ensemble des noeuds définissant l'intersection des plus courts chemins pour chaque joueur.

Data : Tablier du jeu **T**, Couleur du joueur **c**, Profondeur **p**, Fonction d'évaluation **f**
Result : Le coup **n**

```
Chemin1 ← plus court chemin entre les deux bords colorés en c de T ;
Chemin2 ← plus court chemin entre les deux bords colorés en c de T ;
Region ← Chemin1 ∩ Chemin2 ;
if Region est vide then
    n ← coup aléatoire valide ;
else
    n ← Minimax(T, p, −∞, +∞, Region, Minimisation, f)
colorer dans le graphe n choisi par la couleur c ;
retourner n ;
```

Algorithme 5 : Principe algorithmique du coup évalué par Minimax à travers l'évaluation f

La fonction *Minimax* utilisée est optimisée par l'élagage alpha-Beta, ce sont les deux paramètres initialisés lors de l'appel de *Minimax* en $+\infty$ et $-\infty$, le paramètre *Minimisation* est un booléen indiquant le souhait de la minimisation de la distance entre les deux bords pour ce joueur, p c'est la profondeur du calcul.

5 Conclusion

Ce projet avait pour objectif l'implémentation d'un serveur organisant une partie du jeu de Hex entre deux clients. Partant d'une stratégie aléatoire, plusieurs autres stratégies ont étés implémentés dont les plus efficaces sont la stratégie du coup maintenant le plus court chemin et celle des "two-distances".
Ce projet nous a ainsi permis d'approfondir nos connaissances en programmation impérative comme l'utilisation des bibliothèques dynamiques ainsi que leur chargement mais aussi de mettre en application des notions de théorie des graphes comme le parcours en profondeur et le problème du plus court chemin.