
# Rapport de Projet C

*Hex Project*

**Auteurs :** Ibrahim LMOURID & Houssam BAHHOU & Ayoub LASRI & Eloi Magon De La Villehuchet - Equipe $n^o$ 9152
**Encadrants :** Monsieur Frédéric Herbreteau

Première année, Semestre 6,Filière informatique,ENSEIRB-MATMECA
Date : December 12, 2020

# Contents

# 1 Introduction au jeu de Hex

## 1.1 Contexte et présentation du sujet

Le jeu de ***Hex*** a été inventé pour la première fois en 1942 par **Piet Hein**, un scientifique, mathématicien, écrivain et poète danois. En 1948, **John Nash**, un mathématicien américain à Princeton a redécouvert le jeu, qui est devenu populaire parmi les étudiants diplômés à Princeton. Ce jeu fait appel à la logique, à l'image des échecs. Son étude fait partie de la théorie des jeux, et d'autres branches mathématiques.

Hex est joué avec deux personnes, sur un plateau généralement en forme de losange composé de cellules hexagonales. Les dimensions du plateau de jeu peuvent varier, mais la taille typique est de $11 \times 11$. Deux côtés opposés du plateau sont noirs et les deux autres côtés sont blancs (Figure **??**). Au début de la partie le plateau est vide, les joueurs alternent les tours pour placer leur coup sur n'importe quel espace inoccupé sur le plateau de jeu, en lui assignant leur couleur, dans le but de former une chaîne ininterrompue de pions reliant les deux bords. Un résultat non trivial dont la preuve est liée au théorème de Brouwer affirme qu'il existe toujours un vainqueur (*ie* le match nul n'existe pas au Hex).

Figure 1: Un exemple du plateau du jeu hexadécimale

**Convention:** Par convention, dans la suite du rapport, le joueur qui joue avec les pions noires est appelé **BLACK**, celui qui joue avec les pions blanches, **WHITE**.

Une règle supplémentaire, est la règle du gâteau (**pie rule** en anglais). Si le joueur **A** commence, il joue son premier coup noir, l'autre joueur **B** a alors le choix entre jouer à son tour ou intervertir les couleurs. Dans le 2e cas, le joueur qui a commencé devient alors BLACK, et joue alors son premier coup en tant que BLACK, le jeu continuant ensuite normalement. Cela impose de jouer un premier coup ni trop fort (car donnerait un avantage au joueur adverse qui se l'approprierait) ni trop faible (il serait alors laissé par l'adversaire) et réduit l'avantage de commencer.

## 1.2 Problématique

L'objectif du projet était alors d'implémenter une simulation du jeu *Hex* en `langage C`.
Pour ce faire, il a fallu subdiviser le travail de la manière suivante:

- L'implémentation du plateau
- La création d'un ensemble de joueurs (clients)
- L'implémentation d'un serveur organisant le jeu entre deux clients

Le but sous-jacent du projet était de se familiariser avec la manipulation des bibliothèques dynamiques, et d'appliquer les connaissances acquises en théorie des graphes et en programmation impérative.

# 2 Cadre du travail

## 2.1 Outils de travail

Pour pouvoir partager nos avancements nous avons utilisé Git.
Ce logiciel nous a permis d'effectuer un travail collaboratif et de conserver chaque version déposée afin d'éviter les mauvais choix et de pouvoir revenir sur nos pas.
Ce compte rendu est rédigé en LaTeX, les fichiers relatifs à son écriture sont présents sur le dépôt.

## 2.2 Architecture des fichiers et détails techniques

Les fichiers dans le dépôt sont organisés de la manière suivante:

```
/              -- la racine du répertoire du projet
Makefile       -- Makefile global
README.md      -- fichier qui contient les instructions d'exécution
/src           -- fichiers sources (.c/h)
/install       -- Répertoire Install
/rapport       -- Rapport du projet (.tex/pdf)
```

Le fichier `Makefile` fournit :

- **une règle `build`** qui compilera l'ensemble du code,
- **une règle `test`** qui compilera les tests, sans les exécuter
- **une règle `install`** qui copiera les exécutables (server, exécutables de tests alltests*, et un nombre non spécifié de fichiers .so) à l'intérieur du répertoire désigné, et
- **une règle `clean`** qui supprimera les fichiers compilés et installés.

Le `Makefile` prends une variable `GSL_PATH` qui indique le répertoire où est installée la bibliothèque `libgsl.so`.

# 3 Structures implémentées et choix effectués

L'objectif de cette section est de décrire l'ensemble des types abstraits de données et structures implémentées, servant de briques de base à la réalisation du sujet.

## 3.1 TAD `graphe_t`

Le tablier du jeu est en fait un système formé de plusieurs cases connectées pouvant se trouver dans plusieurs états au cours du jeu. Par conséquent l'outil le plus propre pour représenter ce jeu est la théorie des graphes.
En effet le jeu de *Hex* fait partie des jeux à objectif compétitif sur un graphe, dont les règles et les conditions de victoire sont reliées à un problème d'optimisation, l'idée donc du sujet se trouve dans l'association de chaque plateau du jeu à un graphe grille dont les noeuds représentent les différentes cases du tablier.

Le sujet propose de travailler sur trois configurations du plateau avec des tailles variables : Une configuration hexagonale (typique), carrée et triangulaire. Dans notre cas nous avons arrivé à travailler juste sur les deux configurations hexagonale et carrée.

Le TAD `graph_t`, contient une représentation symbolique du plateau qui nous permet de traiter rapidement et efficacement les différentes opérations du jeu sous forme d'opérations sur les graphes.
La structure correspondante à ce TAD a pour champs:

- le nombre de sommets **`num_vertices`** (noté n dans la suite du rapport)
- **La matrice t** qui est une matrice d'adjacence de taille $n \times n$ représentant les arêtes du graphe.

- **La matrice o** qui est une matrice $2 \times n$ codant sur chaque ligne les sommets du graphe appartenant au joueur correspondant (donc 0 ou 1)

Les deux matrices **t et o** sont des matrices creuses, que l'on va définir à l'aide de `GNU Scientific Library`, une bibliothèque `C et C++` permettant de faire des calculs numériques, elle fournit aussi des implémentations et des méthodes de matrices denses et creuses, on verra ultérieurement en détail les méthodes associées au `TAD graph_t`.

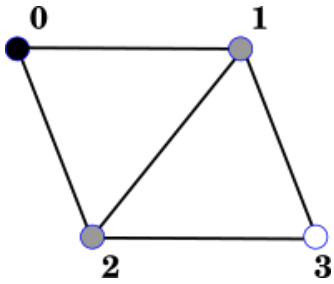


Figure 2: Exemple de graphe hexagonales à 4 sommets

$$\begin{cases} \texttt{num\_vertices} = 4 \\ \texttt{t} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \\ \texttt{o} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{cases}$$

Représentation équivalente du graphe par la structure `graph_t`

### 3.2 TAD `neighbours`

Afin d'évaluer la connexité entre les bords du plateau, ou implémenter des méthodes et des opérations sur le graphe, il nécessite d'abord de pouvoir connaître les voisins de chaque sommet dans le graphe. Pour cela on a implémenté le TAD **`neighbours`**, dont la structure contient les champs suivants:

- **`nb`**: tableau alloué dynamiquement.
- **`size`**: taille de `nb`
- **`capacity`**: capacité de `nb`

Ce TAD contient l'implémentation des méthodes qui servent à extraire les voisins de n'importe quel noeud pour les deux configurations implémentées hexagonale et carrée.

### 3.3 TAD `Path`

Au cours du jeu les deux joueurs essayent de former des chaînes continues afin de connecter les deux bords du plateau, et donc parmi les possibilités qui existent ils cherchent à construire le bon chemin afin de gagner la partie.
Le TAD `Path` permet donc à l'aide de ses méthodes de stocker le chemin choisit sous forme d'un tableau alloué dynamiquement, la structure `Path` est la suivante :

- **`path`**: tableau alloué dynamiquement.
- **`size`**: taille de `path`
- **`capacity`**: capacité de `path`

### 3.4 Autres structures

1. Afin de respecter **la convention** énoncé en Introduction, on a crée un type énuméré **`enum color_t`** qui nous fournit les constantes BLACK=0, WHITE=1, NO_COLOR=2 correspondant au deux joueurs.
2. **`structure move_t`**: pour placer un pion dans une position donnée, il faut d'abord donner une représentation symbolique à ce coup, c'est le rôle de la **`structure move_t`** dont les champs correspondent à:
   - **`m`**: un indice entre 0 et n-1.
   - **`c`**: la couleur du joueur.
3. **`structure player (player-load.h):`** Chaque client est censé être **automatique**, et donc il sera manipulé au cours du jeu par le serveur, à l'aide de plusieurs méthodes, par exemple: initialiser la couleur du joueur, ou bien jouer un coup dans son tour... Afin de donner une représentation symbolique à nos clients , on a choisit donc de créer une structure qui contient les pointeurs des fonctions (méthodes) correspondant à chaque joueur(décrites dans l'interface **`player.h`**), ces pointeurs de fonctions sont :

```
1         struct player {
2          char const* (*get_player_name)();//avoir le nom du joueur
3          struct move_t (*propose_opening)();//proposer le 1er coup
4          int (*accept_opening)(const struct move_t);//accepter ou intervertir le 1er coup
5          void (*initialize_graph)(struct graph_t*);//initialiser le graphe du joueur
6          void (*initialize_color)(enum color_t);//initialiser la couleur du joueur
7          struct move_t (*play)(struct move_t);//jouer le coup
```

```
8           void (*finalize)();//annoncer la fin du jeu
9           void * handle;
10          };
11
```

4. Dans l'interface `.c` de chaque joueur (`player.c , heuristic.c...`) il nous a paru important comme choix de programmation, de regrouper l'ensemble d'informations à propos du joueur (nom, couleur, graphe, nombre de case libre..) sous forme d'une **`structure player`** autre que celle décrite avant.

# 4 Méthodes algorithmiques

## 4.1 Plateau du jeu

L'implémentation d'un plateau du jeu se base sur le TAD `graph_t` décrit dans les fichiers `graph.c, my-graph.c/h` .
En fait, les joueurs possèdent des pions à leur couleur qu'ils disposent tour à tour sur une case de leur choix et un par un. Le tablier se remplit ainsi progressivement. La manipulation de ces actions se fait principalement par les fonctions suivantes:

- **`graph_initialize`**: c'est une fonction de plus haut niveau d'abstraction: prend en argument un entier `size_t n` et un caractère `char c` et initialise un tablier vide de nombre de sommets `n`, et de forme c (hexagonale ou carré) en faisant appel aux fonctions `initialize_carre`, `initialize_hex` et `intialize_o`.

- **`initialize_o`**: puisqu'on a implémenté que les deux configurations carrée et hexagonale, une implémentation commune de la matrice `o` suffit, cette fonction prends la matrice `o` déjà allouée et l'initialise.

- **`initialize_carre/hex`**: allouent et renvoient un graphe carré (respectivement Hexagonale) de `n` sommets après avoir initialisé ses matrices creuses `t` et `o` .

- **`graph_display`** Affiche le graphe donné en paramètre selon sa forme, son résultat est ce qu'on peut voir dans la figure

- **`graph_add_move`**: cette fonction permet d'ajouter le coup donné en paramètre au graphe correspondant

- **`graph_free`**: elle sert à libérer tout espace mémoire alloué dans le graphe (matrices creuses t, o et le graphe lui même)

## 4.2 serveur

Le serveur est principalement responsable de l'arbitrage, de la coordination des parties, de la préparation et la production de résumés des matches. La première tâche du serveur est d'organiser le mode du jeu à l'aide de la fonction `parse_opts`, en utilisant les paramètres données en ligne de commande, qu'on peut résumer de la manière suivante:

- L'option **`-m`** permet de spécifier la largeur du plateau de jeu (ex. **`-m 11`**). (notation valable pour la suite du rapport)
- L'option **`-t`** permet de spécifier la forme du plateau de jeu (grille carrée c ou hexagonale h) (ex. **`-t h`**).
- Les clients sont passés en paramètre dans l'ordre 1er joueur / 2nd joueur

Par conséquent, la ligne de commande suivante permet d'exécuter une partie de Hex en chargeant les deux joueurs `player1.so` et `player2.so` d'une manière dynamique:

```
./install/server -m [M] -t [T] ./install/player1.so ./install/player2.so
```

Ensuite, les deux joueurs chargés sont stocké dans le tableau `players[2]` en utilisant les fonctions `dlopen, dlsym, dlerror et dlclose`, fournies par la Dynamic Loaded (DL) Libraries. Après, le serveur se charge d'initialiser les différents plateaux de jeu correspondant aux serveur, player1 et player2.
Maintenant que tout est prêt, le serveur lance le jeu, le 1er joueur passé en ligne de commande propose le premier coup, puis **la règle de gâteau** (décrite en introduction) intervient pour permettre au serveur de déterminer le joueur **BLACK** et **WHITE**, et la boucle du jeu continue.

Finalement, le serveur arrête le jeu et affiche les résultats de la partie une fois que la fonction `is_winning` lui annonce un gagnant, ou que le plateau est plein. La boucle du jeu correspond au pseudo-algorithme suivant:
Boucle du jeu each player p p→initialize_graph(graph) move = first_player→propose_opening() second_player→accept_o
// 2nd player plays next first_player→initialize_color(BLACK) second_player→initialize_color(WHITE) [//
2nd player refuses and 1st player plays next] first_player→initialize_color(WHITE) second_player→initialize_color(BLACK
true p = compute_next_player() move = p→play(move) the board is full break is_winning() break each player
p p→finalize()

Concernant la fonction **`is_winning`**, son objectif est de vérifier qu'il existe un vainqueur, autrement dit qu'il existe deux cotés opposés reliés, pour cela il faut parcourir la grille en cherchant un chemin connectant ces deux bords. On a choisit donc d'implémenter cette fonction en se basant sur **un algorithme de recherche en profondeur**.

**L'algorithme** commence au nœud racine (*i.e.* **0** dans le cas d'un joueur BLACK et **m+1** pour le joueur WHITE) et explore autant que possible le long de chaque branche avant de revenir en arrière. L'idée de base est donc de partir de la racine, de marquer le nœud et de se déplacer vers le nœud adjacent non marqué et de continuer cette boucle jusqu'à ce qu'il n'y ait pas de nœud adjacent non marqué ou notre algorithme atteint le coté opposé, si on est dans le deuxième cas (**condition de gain**) on annonce qu'il y a un vainqueur en retournant 1, sinon on revient ensuite en arrière, on vérifie les autres nœuds non marqués et on les parcoure, finalement si notre algorithme retourne 0 et non 1 on est donc dans le cas d'**un match nul dans un plateau carré**.

## 4.3 Clients

## 4.4 Le Random: `Hero.so (player.c)`

Le joueur **aléatoire** correspond à une version plus naïve d'un joueur. En fait, ce joueur commence par sauvegarder le coup de son adversaire dans son plateau du jeu, ensuite il choisit une case d'une manière aléatoire, tant que la case choisie n'est pas vide, le joueur continue ses tentatives aléatoires, lorsque cette case est libre, le joueur place son pion dans celle ci

## 4.5 `Bridge.so`

Ce joueur maintient une stratégie défensive, qui lui permet de gagner devant des joueurs aléatoires, en fait cette stratégie n'est pas basée sur une évaluation intelligente, mais essaie d'empêcher l'adversaire de connecter ses bords.
En effet, à partir de la position de l'adversaire ce joueur essaie un bloc en mettant son pion à une distance de deux chaînes du pion de l'adversaire. C'est mieux que le bloc adjacent, mais parfois l'adversaire peut également contourner cela en enchaînant deux en biais. Si ce coup est impossible à jouer, `bridge` essaye de jouer un bloc adjacent ou un coup aléatoire valide. La figure ci dessous illustre un bloc adjacent (1) et un bloc à deux chaînes (3):

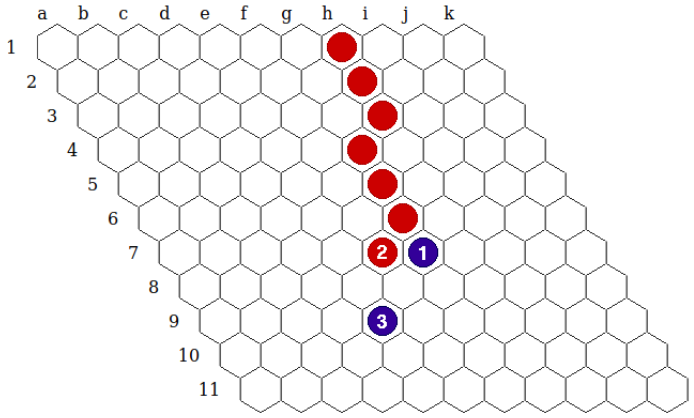


Figure 3: Illustration de la stratégie de bridge

## 4.6 Les heuristiques

Afin de gagner des parties de Hex il faut penser à créer des joueurs intelligents capables de rivaliser avec succès à l'aide des stratégies gagnantes.
Un joueur possède une stratégie gagnante pour un jeu s'il peut s'assurer la victoire quels que soient les coups joués par son adversaire. Par conséquent le but de cette section est d'implémenter des algorithmes décrivant

ces stratégies.

D'abord, un paradigme important, non seulement pour les jeux compétitifs à deux joueurs (Hex, échecs, Tic Tac Toe...) mais pour la programmation de jeux en général, consiste en des méthodes et des algorithmes évaluant autant de mouvements que possible jusqu'à l'expiration du temps de réflexion.

Plusieurs de ces des algorithmes de recherche ont été développés au cours des dernières années, comme **Minmax (Shannon, 1950),** $(\alpha - \beta)$ **Pruning (Knuth et Moore, 1975)** qui sera la base de nos joueurs intelligents.

### 4.6.1 Minmax $(\alpha - \beta)$

Le but de cet algorithme est de produire un nombre lié à la possibilité d'une victoire compte tenu de la position actuelle du pion, en essayant de maximiser les gains pour un joueur et les minimiser pour l'autre.
Étant donné une position, on suppose que c'est au joueur BLACK de jouer. Dans la position donnée, BLACK a une série de coups qu'il peut effectuer : pour chacun d'eux, il s'interroge sur les répliques éventuelles que peut faire WHITE, qui lui-même analyse pour chacune de ses répliques celles auxquelles peut procéder BLACK, qui à son tour examine à nouveau l'ensemble des coups qu'il peut effectuer suite aux répliques de WHITE, etc.

Ce processus de réflexion est généralement associé à **un arbre** de jeu. Chaque nœud y correspond à une position de jeu et les branches correspondent aux différents coups que peut faire BLACK ou WHITE à partir de cette position.
Cependant le plateau de jeu contient un nombre important de cases, cela fait de très nombreuses combinaisons à comparer, par conséquent il sera impossible d'atteindre des positions terminales. Pour cela, il est nécessaire de mettre en œuvre la stratégie MinMax en utilisant **une profondeur** d'arbre prédéfinie limitant la recherche; les feuilles de l'arbre ne sont donc pas des positions de fin de jeu, ainsi il n'est pas possible de dire si une feuille correspond à une position gagnante ou à une position perdante. Pour résoudre ce problème, il est ainsi nécessaire de disposer **d'une fonction d'évaluation (heuristique)**, capable d'estimer le plus précisément possible la qualité d'une position (maximale pour un joueur et minimale pour l'autre ou l'inverse), et qui servira à étiqueter les feuilles de l'arbre par des valeurs numériques. Par la suite on aura deux types de noeuds : un nœud MAX choisit un coup de score maximal parmi ses fils et un nœud MIN choisit un coup de score minimal parmi ses fils.
Le nombre de branches que cet algorithme de **MinMax simple** amène à développer est extrêmement important, soit **b** la valeur moyenne de ce nombre et **d** la profondeur ce cette arbre, la complexité de cet algorithme récursif est $\mathbf{O}(b^d)$, cela implique donc que la fonction d'évaluation doit être appliquée à $b^d$ cases, ainsi la recherche d'un coup va prendra des minutes!!

Une amélioration de cet algorithme appelée **Élagage Alpha-Beta** permet d'éviter de développer des branches inutiles: si la valeur d'un fils $f$ d'un noeud MAX est supérieure à la valeur courante d'un noeud MIN ancêtre, alors les frères de $f$ n'ont pas besoin d'être explorés, ce principe s'appelle **Coupure Beta**.
La figure suivante est un exemple illustrant l'arbre Minmax AlphaBeta:

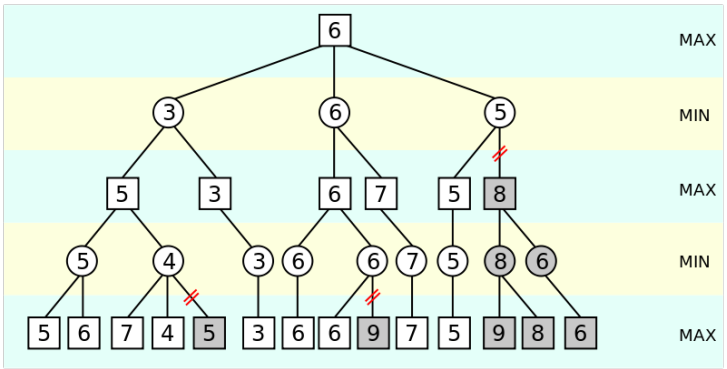


Figure 4: Exemple d'arbe de Minmax AlphaBeta

Ce processus est décrit par le pseudo-code suivant: