Projet GO – RAPPORT

- **Structures de données**
  - Structures :

```
/**
 * repésente une coordonnée avec position en x et posotion en y
 * x,y : les positions x et y
 */
typedef struct sCoord
{
    int posX;
    int posY;
} Coord;
```

→ Structure servant à maintes reprises à stocker une coordonnée, c'est-à-dire son abscisse et son ordonnée. Notamment pour situer les intersections du goban.

```
/**
 * représente une intersection du goban (plateau)
 * position : Coord (x,y)
 * nbLibertes : nombre de libertes de l'intersection
 * estOccupe : si une pierre est posée sur l'intersection ou non
 * couleur : couleur de la pierre si l'intersection est occupée
 * type : Correspond au type d'emplacement de l'intersection (ex : coin haut)
 * suiteChaine : Intersection suivante dans la chaine
 * chMere : chaine à laquelle appartient l'intersection
 */
typedef struct sIntersection
{
    Coord* position;
    int nbLibertes;
    bool estOccupe;
    CouleurPierre couleur;
    TypeInter type;
    struct sIntersection* suiteChaine;
    struct sChaine* chMere;
} Intersection;
```

→ Structure représentant une intersection du goban, c'est-à-dire les emplacements de pose des pierres. Dans le programme, une intersection contiendra ses coordonnées, son nombre de libertés en temps réel, sa situation (occupée ou non par une pierre, sa couleur) ainsi que son type (coin, bord, etc.). On souhaite également conserver une référence vers sa « chaîne mère » si il y a lieu. Dans la même optique de chaînage, une intersection pointera ou non vers une autre (à la manière d'une liste chaînée).

```
/**
 * contient toutes les intersections d'une chaîne
 * nbPierres : le nombre de pierres dans la chaine
 * debutChaine : La première intersection de la chaine
 * finChaine : La dernière intersection de la chaine
 */
typedef struct sChaine
{
    int nbPierres;
    Intersection* debutChaine;
    Intersection* finChaine;
} Chaine;
```

→ Structure devant représenter une chaîne de pierres de même couleur sur le goban. Dans le cadre du programme, on souhaite reproduire un modèle de liste chaînée et une Chaine contiendra donc un

pointeur vers une intersection représentant son premier élément, ainsi qu'un pointeur vers une intersection représentant son dernier élément. Permettant respectivement d'accéder au début de la chaîne afin de la parcourir et d'accéder à la fin de la chaîne quand besoin (sans devoir forcément la parcourir d'élément en élément si seul le dernier élément est requis).

- Énumérations :

```
/**
 * énumération des couleurs de pierre
 * BLANC : pierre blanche
 * NOIR : pierre Noire
 */
typedef enum eCouleurPierre
{
    BLANC = 1,
    NOIR = 2
} CouleurPierre;
```

→ Énumération des différentes couleurs de pierre possibles. La valeur par défaut d'une variable de type « énumération » étant l'entier 0, on attribue ici les valeurs 1 à BLANC et 2 à NOIR afin de pouvoir tester l'absence de couleur d'une intersection facilement. (intersection → couleur != 0)

```
/**
 *  énumération des différents types d'intersections (en fonction de leur position)
 *  COIN_HG, COIN_HD, COIN_BG, COIN_BD : respectivement les quatres coins du plateau
 *  BORD_HAUT, BORD_DROIT, BORD_BAS, BORD_GAUCHE : les quatres bords du plateau
 *  DEFAUT : les autres intersections
 */
typedef enum eTypeInter
{
    COIN_HG,
    COIN_HD,
    COIN_BG,
    COIN_BD,
    BORD_HAUT,
    BORD_DROIT,
    BORD_BAS,
    BORD_GAUCHE,
    DEFAUT
} TypeInter;
```

→ Énumération des différents types d'intersection du goban possibles. Permettra entre autre au sein du programme de déterminer facilement quels sont les intersections adjacentes ou plus simplement combien il y en a au total.

```
/**
 * énumération qui indique le mode de jeu
 * MENU : mode de choix du plateau
 * JEU : mode de jeu
 */
typedef enum eMode
{
    MENU,
    JEU
} Mode;
```

→ Énumération des deux modes du jeu possibles, c'est-à-dire (dans l'état actuel du programme), soit le menu de choix des plateaux, soit les plateaux. Permet notamment, au clic de la souris, de déterminer la suite d'actions à réaliser.

- Liste Chaînée :

Dans le cadre de ce programme, afin de pouvoir implémenter la fonctionnalité de chaînage des pierres du jeu de go, une structure de donnée de type liste chaînée est utilisée. (voir structures « Intersection » et « Chaine »)

Une liste chaînée désigne en informatique une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments de même type, dont la représentation en mémoire de l'ordinateur est une succession de cellules faites d'un contenu et d'un pointeur vers une autre cellule. De façon imagée, l'ensemble des cellules ressemble à une chaîne dont les maillons seraient les cellules.

L'accès aux éléments d'une liste se fait de manière séquentielle : chaque élément permet l'accès au suivant (contrairement au tableau dans lequel l'accès se fait de manière directe, par adressage de chaque cellule dudit tableau).

(source : https://fr.wikipedia.org/wiki/Liste_cha%C3%AEn%C3%A9e)

- Tableaux :

Le programme utilise des tableaux notamment dans les cas suivants :

- Tableau lesInters : variable static. Contient toutes les intersections du goban. Initialisé et instancié au choix du plateau.
- Tableau lesChaines : variable static. Contient toutes les chaînes du goban au fur et à mesure de la partie, c'est-à-dire avec ajouts et suppressions d'éléments en fonction des actions de la partie.
- Tableau lesAdjacents : retourné par la méthode getLesAdjacents(), contient les pointeurs des intersections adjacentes à une intersection donnée.
- Tableau lesLibertes : retourné par la méthode getLesLibertés, contient les pointeur des intersections représentant les libertés d'une intersection donnée à un temps de la partie.

- **Structure du programme**
  - Le code source du programme a été réparti dans les fichiers suivants :
    - dessine.c (fourni par l'enseignant) : fonctions représentant la logique graphique du programme. (voir libraire X11)
    - dessine.h (fourni par l'enseignant) : fichier header contenant les définitions de fonctions de dessine.c (voir libraire X11)
    - jeu.c : ensemble des fonctions représentant la logique métier du programme (ligne de conduite). Contient également la méthode main permettant de lancer celui-ci.
    - Jeu.h : fichier header contenant les définitions de fonctions, structures, énumérations et macros du fichier jeu.c
  - On notera également la présence d'un Makefile contenant l'intégralité des instructions de compilation.

- **Choix techniques effectués**
  - Le programme impliquant de pouvoir modifier les données de certaines des structures par l'appel de méthodes, il a donc été décidé de construire la logique de celui-ci autour de l'utilisation de pointeurs (avec allocations dynamiques de mémoire et free()).
  - Pour éviter des erreurs dues à la saisie utilisateur lors du choix du plateau, une interface de choix des plateaux à la souris (boutons) a été implémentée.