Projet de GO

Organisation

- Utilisation de git

L'utilisation de git[1] nous a permit une bonne gestion et une bonne organisation de notre travail étant donné que ce logiciel permet l'upload de version à chacunes des modifications effectuées par l'un de nous deux, et la determination de conflits s'il y en a. Commandes :

```
ssh-add FILE.key
```

Remarque: ne s'utilise qu'une seule fois pour l'ajout d'un nouveau fichier/dossier sur le dépôt[2].

Mise à jour du repertoire[3]

```
git pull origin master
```

Ajout d'un nouveau fichier/dossier

```
git add FILE
```

Dire ce que l'on a fait sur les modifications courrantes

```
git commit -a
```

IDE utilisé par les deux membres du binôme : GEANY qui est un editeur equivalent à CodeBlocks et très pratique pour la réalisation de projets.

- Organisation du dossier pour le projet

Etape 1: (de la réalisation du projet GO) c'est l'organisation des dossiers.

```
.
├── bin
│   ├── go
│   ├── test_capture_chaines
│   ├── test_plateau
```

```
│   └── test_territoire
├── doc
├── doxygen_config
├── doxygen_config.bak
├── extra
│   ├── ensi_logo.jpeg
│   ├── plateau_initiale_13_13.txt
│   ├── plateau_initiale_19_19.txt
│   ├── plateau_initiale_9_9.txt
│   ├── plateau_test_13.txt
│   ├── plateau_test_capture_chaines.txt
│   ├── plateau_test_output.txt
│   ├── plateau_test_territoire.txt
│   ├── plateau_test.txt
│   └── sauvegardes
│       ├── 1
│       ├── 2
│       ├── 3
│       ├── 4
│       ├── 5
│       ├── 6
│       ├── 7
│       ├── 8
│       ├── 9
│       └── sauvegarde_partie
├── include
│   ├── chaine.h
│   ├── chaines.h
│   ├── couleur.h
│   ├── ensemble_colores.h
│   ├── ensemble.h
│   ├── ensemble_positions.h
│   ├── libertes.h
│   ├── matrice.h
│   ├── partie.h
│   ├── pion.h
│   ├── pions.h
│   ├── plateau.h
│   ├── position.h
│   ├── positions.h
│   ├── territoire.h
│   └── territoires.h
├── lib
│   ├── ensemble
│   │   ├── bin
│   │   │   ├── libensemble.so
│   │   │   └── test_libensemble
│   │   ├── src
│   │   │   └── ensemble.c
│   │   └── test
│   │       └── test_ensemble.c
```

```
│   └── matrice
│       ├── bin
│       │   ├── libmatrice.so
│       │   └── test_libmatrice
│       ├── src
│       │   └── matrice.c
│       └── test
│           └── test_matrice.c
├── makefile
├── object
│   ├── ensemble_colores.o
│   ├── ensemble_positions.o
│   ├── go.o
│   ├── libensemble.o
│   ├── libertes.o
│   ├── libmatrice.o
│   ├── partie.o
│   ├── plateau.o
│   ├── position.o
│   ├── territoire.o
│   ├── test_capture_chaines.o
│   ├── test_libensemble.o
│   ├── test_libmatrice.o
│   ├── test_plateau.o
│   └── test_territoire.o
├── README
├── src
│   ├── ensemble_colores.c
│   ├── ensemble_positions.c
│   ├── go.c
│   ├── libertes.c
│   ├── partie.c
│   ├── plateau.c
│   ├── position.c
│   └── territoire.c
└── test
    ├── test_capture_chaines.c
    ├── test_ensemble_colores.c
    ├── test_ensemble_positions.c
    ├── test_plateau.c
    ├── test_territoire.c
    └── tmp.c
```

On separe bien les librairies, le code source permettant les tests, et le code source de notre projet. Un makefile, ainsi qu'une documentation est disponible (en html, avec man, etc...).

- Création des .h (écriture des signatures)

Etape 2 : Création des fichiers headers .h conformement à ce qui a été demandé dans la documentation du projet, en plus de la création de la documentation doxygen pour chaque fonction ou type existant dans les fichiers .h. Tous les .h se trouve dans un seul et même dossier include pour faciliter l'étape de compilation. L'utilisation de @bug et @todo nous a été aussi très utile pour revoir ce qui posait problème et sur quoi on devait encore s'attarder dans la phase deboggage.

Les types utilisés pour la réalisation de ce projet sont :

- le type **Pion** qui n'est rien d'autre qu'une position en plus de la couleur de cette position.

- le type **Ensemble** qu'on a codé sous forme d'une liste chainée.

```
struct Ensemble{
    Cell* courant;
    Cell* tete;
};
```

Remarque : La notion d'ordre n'apparait pas dans notre cas, malgré qu'on manipule des listes chainées qui respectent l'ordre de positionnement. Ici dans notre cas il s'agit que d'une manipulation de positions l'une après les autres pour former soit des ensembles de positions (cf **Ensemble_Positions**) soit des ensembles de positions de couleur NOIR/BLANCHE/VIDE. On definit donc le type **Couleur** qui est un type énuméré.

La création du type **Ensemble** nous a servi de support pour la création de plusieurs autres types en reprenant exactement les mêmes fonctions de ce type comme **Ensemble_Positions** ou **Ensemble_Colores**. D'autant plus que tous les autres types proposés ne sont que des alias des types **Ensemble**,**Ensemble_Positions** et **Ensemble_Colores**. C'est donc les types les plus fondamentaux et les plus importants.

-Le type Matrice qui est aussi un type essentiel pour la création du plateau. C'est un type qui permet de manipuler les données du plateau de jeu, ainsi que la taille du plateau (nbr de lignes et nbr de colonnes).

```
typedef struct Matrice {
    int **donnees; /** Les tableau des données */
    int nbligne; /** Le nombre de lignes */
    int nbcolonne; /** Le nombre de colonnes */
} Matrice;
```

-Le type plateau n'est rien d'autre qu'un alias du type Matrice. En effet, le plateau du jeu est une matrice de taille nxn avec n=9 ou n=13 ou n=19, comportant des pions qui ne sont rien d'autres que des positions dans la matrice. Ils sont donc repérés par leur coordonnées dans cette matrice.

Après avoir défini les types, on a créé les prototypes de chaque fonction associée à chaque type. Par exemple, le type **Ensemble** lui sont associées les fonctions :

```
void ensemble_init(Ensemble *E);
int ensemble_vide(Ensemble* E);
void ensemble_ajouter(Ensemble* E, void* element);
Ensemble* ensemble_enlever(Ensemble* E, void* element);
Cell* ensemble_tete(Ensemble* E);
```

```
Cell* ensemble_courant(Ensemble* E);
int ensemble_suivant(Ensemble* E);
```

NB : Notez que les noms de nos fonctions ne sont pas aléatoires mais sont choisis expres pour faciliter la manipulation de toutes ces fonctions. Chaque nom de fonction commence par ensemble_xxx et ce pour obtenir toutes les fonctions à l'aide de la touche TAB (tabulation) qui permet de retrouver toutes ces fonctions dans l'editeur utilisé. Exemple : Geany/Code Blocks ... Même chose pour les fonctions des autres types. C'est une manière de faciliter la manipulation de plusieures fonctions sans se perdre.

Nous avons pensé à créer une fonction:

```
 void ensemble_detruire( **Ensemble* ** E );
```

qui permet la “destruction” d'un ensemble passé en paramètre, ou ce qu'on appelle libération de l'espace memoire alloué pour cet ensemble, mais nous ne l'avons utilisé à aucun moment, vu qu'il y avait surtout des modifications à faire sur chaque étape du jeu sur le même ensemble plutôt que des changements d'ensemble qui necessitent la supression et le renouvellement de nouveaux ensembles.

- Ecriture des fonctions

Chaque membre du binôme s'est chargé de réalisé une fonction. La créaction d'un type **Ensemble** nous a servi de support pour la création de tous les autres types notament **Ensemble_Colores** et **Ensemble_Positions** qui utilisent les mêmes fonctions seulement le contenu de chaque structure change. Le pointeur generique void* nous a permit de gerer la modification des contenus specifiques à chaque nouveau type. Nous avons donc procéder de manière organisée en faisant des cast et ce qu'on appelle aussi de l'**encapsulation** (terme valable en JAVA, signifie la modification des fonctions déjà codées pour les adapter aux nouveaux types, en plus de l'ajout de fonctions spécifiques uniquement pour le type en question). Exemple :

```
int ensemble_colores_vide(Ensemble_Colores* E)
{
    return ensemble_vide(E->p);
}
```

- Création du makefile

Le makefile a été modifié au fur et à mesure de la creation de fonctions et de fichiers sources. Plusieurs regles ont été crées selon ce qu'il fallait compiler. Les definitions des macros a suivi l'organisation choisie de nos dossiers :

```
LD_LIBRARY_PATH=$(shell pwd)

INCDIR=./include
BINDIR=./bin
LIBDIR=./lib
```

```
OBJDIR=./object
SRCDIR=./src
TESTDIR= ./test
```

Definition des FLAGS : un pour le linkage des .o et compilation pour le .out (executable)

```
CFLAGS=-I $(INCDIR)
```

un pour la compilation des librairies dynamiques

```
LDFLAGS=-L $(LIBDIR)/ensemble/bin -L $(LIBDIR)/matrice/bin \
        -Wl,-rpath,$(LD_LIBRARY_PATH)/lib/ensemble/bin \
        -Wl,-rpath,$(LD_LIBRARY_PATH)/lib/matrice/bin
```

Remarque : L'utilisation des librairies a été très pratique car elle nous a faciliter la manière de gérer le nombre de fonctions à utiliser. Nous avons pensé à la création de librairies dynamiques plutôt que statiques étant donné que c'est des fonctions pouvant être utilisées pour d'autres projets ultérieurs. Il faut noter aussi que nous avons repris une librairie dynamique déjà codée auparavant qui est la libmatrice du jeu de vie (ODL/TP4) qui regroupe toutes les fonctions nécessaires pour la création d'un plateau, la sauvegarde, le chargement du plateau etc. D'où l'efficacité des librairies dynamiques.

Lors de l'utilisation de la librairie, on ajoute la dependance de la librairie en question pour qu'elle soit mise à jour s'il y a une modification et pour que le linkage se fasse ensuite proprement: (ajout de l'option -lnamelibrairie lors de la compilation)

Test fonctions

Apres l'étape du makefile, il faut tester les fonctions, et par la même occasion le makefile. Un dossier Test a été créé pour ce faire. Il contient plusieurs fichiers test qui permettent de tester le bon fonctionnement des fonctions utilisées dans le codage du jeu au fur et à mesure qu'elles sont codées.

Difficultés

-La comprenhension du jeu n'a pas été très rapide, il nous a fallu y jouer plusieurs fois pour arriver à comprendre comment on allait s'y prendre algorithmiquement. Même si on a été partiellement guidé à travers un document fournit pour le projet, il est bon d'avoir des exemples pratiques permettant de comprendre les différentes structures, et comment elles doivent être implémentées.

-Ecrire des fonctions particulières[4] en utilisant les connaissances du cours. C'était le cas des ensembles, qui peuvent contenir tout et n'importe quoi [5].

-Suite à notre implication[6] dans la campagne du BDE, il y a eu un manque de temps dans la réalisation complète [7] du projet.

Amélioration à réaliser

- Utilisation de free(), et désalocation mémoire des différents noeuds réalisés lors des ajouts.
- On pourrait se pencher sur l'optimisation de l'algorithme [8] ? mais est-ce que ca vaut vraiment le coup ? sachant qu'on travail sur une matrice 19×19 au maximum
- Mettre les structure dans les .c, mais pose certains problème que je n'ai pas eu le temps d'exploiter [9].

[1] l'un des plus populaire gestionnaire de versions
[2] Le dépôt c'est le dossier distant qui contient tout le projet, et à travers lequel tout les développeurs qui ont accès vont récupérer les fichiers
[3] récupération des dernières modifications à distances
[4] sur mesure
[5] void *
[6] deux membres du binome
[7] utilisation graphique de la SDL
[8] en réduisant par exemple les noeuds Position qui sont parfois identiques dans différents ensembles. On pourrait alors se contenter d'avoir le même dans les différents ensembles en partageant son adresse mémoire
[9] par exemple le contenu de la structure qui n'est pas connue pour certaine fonction qui incluent juste le fichier .h et qui du coup donnent des erreurs du type : incompatible type