

RAPPORT DE PROJET C

1 - Normes	2
2 - Gitlab	2
Les branches :	3
3 - Architecture du projet	3
4 - Interface graphique	4
A – Affichage de SMap	4
B – Gestion des events	6
C – Ajout des « plus »	6
5 - Logs	6
6 - Comportement de l'IA	7
7 - Vérification de la mémoire avec Valgrind	8
8 - Débuggage avec GDB	9

MARAVAL Nathan
MONVOISIN Mathilde
PAGANO Lucas
LE PRIOL Yoann

1 - Normes

Tout le code est écrit en anglais : nom des fonctions et des variables. Les commentaires par contre sont écrits en français.

Le nom des fonctions et des variables sont en minuscules, avec des majuscules pour remplacer les espaces (ex: ceciEstUneVariable). Par contre les macros de Define sont écrits tout en majuscule.


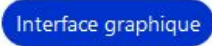


Pour commenter les fonctions nous avons mis en place une petite norme à respecter : une ligne pour dire ce que la fonction fait, puis une ligne par variable en entrée, et enfin une ligne pour dire ce que la fonction retourne :

```
//Fonction qui est un exemple pour la démonstration
//var1 : un entier
//retourne 1 si tout c'est bien passé, 0 sinon.
int exemple(int var1) { return 1; }
```

2 - Gitlab

Les issues :

Pour se répartir les tâches à faire sur le projet, nous avons utilisé le système d'issue de gitlab. Nous les avons classées dans 4 catégories différentes :

☆ 	<u>To Do</u> : les tâches à faire, mais personne ne travaille dessus actuellement
☆ 	<u>IA</u> : Les tâches en rapport avec l'IA, dont au moins une personne travaille dessus actuellement.
☆ 	<u>Interface graphique</u> : Le même type de catégorie que "IA" : mais en rapport avec l'interface graphique.
☆ 	<u>Doing</u> : Les autres tâches en cours, mais sur des sujets annexes.

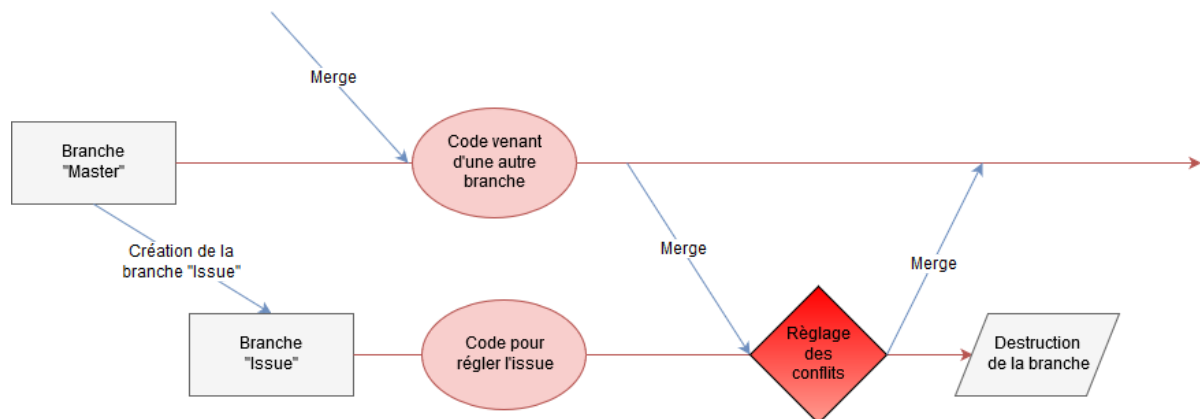
Quand une issue est finie, elle est passée dans l'état Closed. Donc quand une issue est créée, elle est classée par défaut dans To Do en attendant que quelqu'un se l'assigne et la place dans la bonne catégorie.

Les branches :

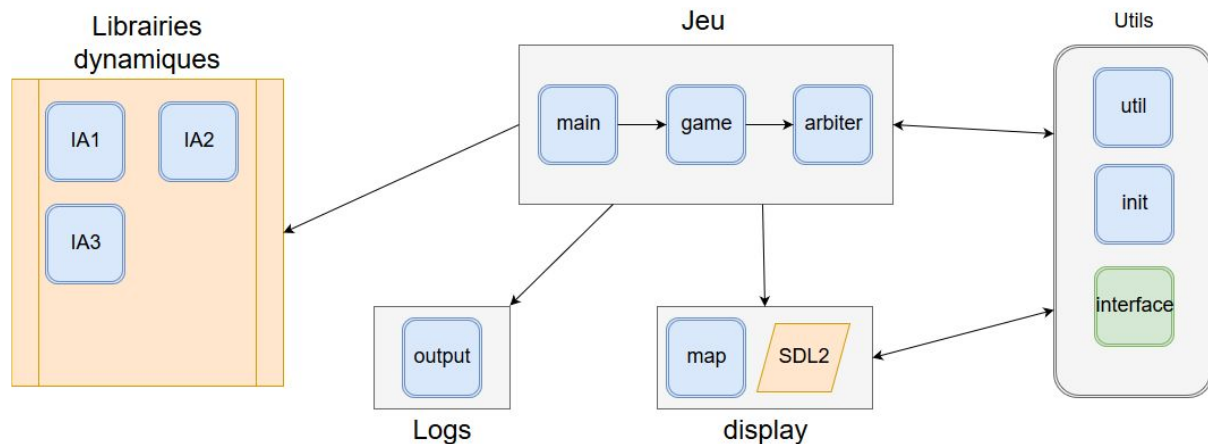
Lorsqu'un membre du groupe s'occupe d'une tâche, il doit éviter de travailler directement sur la branche master. Sinon il risque de mettre en ligne un code non-fonctionnel, ce qui va grandement gêner les autres développeurs. Et de toute façon, tout le monde va finir par se marcher sur les pieds, et devoir régler des conflits tout le temps.

C'est pourquoi pour chaque tâche une nouvelle branche est créée. Ainsi, plusieurs branches peuvent avancer en parallèles sans se gêner. Une fois la tâche finie, la branche master est "mêrgée" à la branche secondaire, afin de régler les conflits toujours hors de la branche master. Une fois la branche mise à jour, il suffit de vérifier les bugs (notamment les pertes mémoires. Enfin, la branche de l'issue est "mêrgée" sur master afin de mettre à jour le code pour tout le monde.

Exemple avec la branche "Issue" :



3 - Architecture du projet



Nous avons répartis les fichiers du projet en fonction de leurs utilité.

Jeu :

Contient les fichiers faisant tourner la boucle de jeu principal. Main va servir à lancer les différentes parties les unes à la suite des autres. Pour cela il va faire appel au fichier game qui va servir à faire le déroulement d'une partie dans son ensemble. C'est aussi dans le fichier game que les logs et l'affichage de l'interface est géré. Enfin arbitrer va contenir les fonctions permettant de gérer les coups des joueurs.

Display :

Ce dossier contient tous les fichiers en rapport avec l'interface graphique. Map va contenir les fonctions permettant de créer la surface, de la rafraîchir ou encore de la détruire. Tout cela en faisant appel à SDL2 évidemment.

Logs :

Fichier permettant la création et l'écriture dans le fichier de logs.

Utils :

Dossier contenant des fonctions générales, susceptibles d'être appelées dans tous les autres fichiers. Util va contenir les méthodes générales, tels que la gestion de l'aléatoire. Init va contenir toutes les fonctions utilisés lors du lancement d'une partie. Et enfin Interface.h contient des structures utiles dans tout le reste du code.

IA :

Les ias vont aussi avoir accès un fichier utilA susceptible d'être utilisé par toutes les ias.

4 - Interface graphique

A – Affichage de SMap

La première question qui nous est venue est : comment générer les cellules et les afficher ? Nous avons donc commencé par créer une fonction qui génère des centres (structure ayant pour attributs une SCell *, un x et un y) aléatoirement.

```
typedef struct Centre{  
  
    SCell *cell;  
    int x;  
    int y;  
}Centre;
```

C'est ensuite la fonction getCloser() qui associait ces centres à tous les autres pixels, en parcourant la SMap * et en associant chaque pixel au centre qui lui était le plus proche. Une autre fonction parcourait de nouveau pour colorer les pixels en fonction de l'id de leur owner. Deux autres fonctions à la structure similaire passaient ensuite pour dessiner un bord noir autour des cellules. Le but était alors d'avoir un affichage de la SMap le plus rapide possible pour pouvoir commencer les étapes plus compliquées.

Pour simplifier le stockage et l'accès à la liste des cellules et l'initialisation de la SMap *, une nouvelle structure a vu le jour :

```
typedef struct MapContext{  
    SMap *map;  
    Centre* cellsList;  
    unsigned int nbNodes;  
} MapContext;
```

Plus tard, il est apparu que notre jeu était lent et nous avons donc pris la décision de changer la structure de cet affichage pour optimiser son exécution. La solution qui a émergé de cette réflexion est qu'il serait préférable de stocker un tableau de pointeurs que de recalculer à chaque tour quel pixel est associé à quel centre, puis quel pixel à quel owner, car cela demanderait beaucoup moins de calculs. C'est donc ainsi que nous avons créé le type Graph[][] qui contient des pointeurs vers des SCell. Ce graph n'a besoin d'être initialisé une seule fois, au tout début de la partie. Il est stocké dans notre MapContext :

```
typedef struct MapContext{  
    SMap *map;  
    Centre* cellsList;  
    unsigned int nbNodes;  
    Graph graph;  
} MapContext;
```

Le graph est actualisé à chaque tour de jeu de l'IA, après la copie de la map.

Un peu plus tard dans notre développement, nous avons ajouté l'affichage d'un dé sur chaque Centre. Le nombre affiché sur ce dernier permet au joueur de connaître le nombre de dés de chaque cellule.

B – Gestion des events

Une fois que l’affichage de la map et toutes les initialisations intrinsèques, il s’agit de gérer les différentes interactions d’un joueur avec le jeu, notamment le clic sur un territoire pour que ces derniers puissent être conquis par les différents joueurs. Cette action se fait assez simplement en récupérant l’id de la cellule sur laquelle le joueur clic en premier. La boucle de jeu vérifie alors si le clic est bien fait sur la map, puis que l’id de la cellule sur laquelle il a cliqué lui appartient, enfin que la cellule contient plus d’un dé, et il attend de recevoir le deuxième clic pour savoir quelle est la cellule à attaquer. Sont alors effectuées 3 nouvelles vérifications ; celles que la cellule à attaquer 1- n’appartient pas à ce même joueur, 2- fait bien partie des voisins (neighbors) de la première cellule, et enfin 3- n’appartient pas à ce même joueur.

Nous avons également choisi, pour pouvoir terminer un tour, de rajouter un bouton « end turn ».

C – Ajout des « plus »

Nous avons choisi d’indiquer, à chaque tour, à qui est le tour à l’aide d’un gros dé de la couleur qui correspond à l’id du joueur dont c’est le tour.

Nous avons également estimé qu’il serait agréable, pour le joueur, de connaître le nombre de dés présents dans son stack. Ce dernier est donc présent à côté du dé indiquant le tour du joueur.

Enfin, nous avons rajouté un titre à notre interface, ainsi que le dé-logo, qui finalise ainsi notre interface.

Dû à des problèmes de merge finaux, ces fonctionnalités n’ont pas pu être implémentées.

5 - Logs

Un fichier externe va récupérer les résultats de la partie. Le but étant de pouvoir refaire le déroulement de la partie à partir de ce fichier texte. Le fichier est généré pour une exécution du programme. C'est à dire qu'une nouvelle exécution va effacer le précédent fichier de logs. Cependant si plusieurs parties sont lancées avec la même exécution du programme, alors les résultats des parties vont être mis les uns à la suite des autres dans le fichier.

Le fichier de log est divisé en plusieurs parties. Chaque partie a une balise propre, et cette balise va être mise devant chaque ligne de la partie. C'est avec ces balises qu'un parseur va pouvoir lire le fichier. Dans chaque ligne, les informations sont séparées par des espaces simples.

Balise player :

Le log commence par les présentations des IAs. Les informations affichées sont celles données par l'IA lors du premier appel de "InitGame". Si la partie n'a pas d'IA, mais uniquement des joueurs humains, cette partie est vide.

Informations affichées :

(entier positif) id de joueur

(entier positif) la taille du nom *puis* le nom

6x : (entier positif) la taille de la string *puis* (string) le nom du membre de l'équipe

Remarque : c'est la seule section qui ne va apparaître qu'une fois dans le fichier, même si plusieurs parties sont lancées dans la même exécution.

Balise cell :

Cette section va donner l'état initial de la map. Le but étant de pouvoir reconstruire une map uniquement à partir de ces informations. Chaque ligne va donner les informations propres à une cellule.

Informations affichées :

(entier positif) id de la cellule

(entier positif) id du player propriétaire de la cellule

(entier positif) nombre de dés initial

(entier positif) nombre de voisins *puis* Nx : l'id d'un voisin

Balise turn :

Cette section va indiquer le déroulement de la partie. C'est à dire tous les coups joués par les player. Chaque ligne correspond à un coup joué.

Informations affichées :

(entier positif) id de la cellule attaquante

(entier positif) id de la cellule attaquée

(entier positif) 1 si l'attaque a gagné, 0 si c'est le défenseur

Balise victory :

Indique le gagnant de la partie.
Informations affichées : (entier positif) id du gagnant

6 - L'IA

A - Le comportement de l'IA

L'IA que nous avons codée se base sur trois grands principes : La probabilité de gagner un coup et le calcul de ses composantes connexes. En utilisant ces principes, elle classe les coups possibles en trois catégories : Les coups non-jouables, qui ne sont pas stockés, les coups jouables, stockés dans `playableTurns`, et les meilleurs coups, stockés dans `bestTurns`.

Les coups non-jouables sont ceux qui ne sont pas permis et ceux pour lesquels on a moins de cellules que l'adversaire. Nous avons fait le choix d'éliminer ces derniers car la probabilité de gagner est trop faible même s'ils pourraient nous apporter un bénéfice important.

Les coups jouables représentent le reste.

Les `bestTurns` sont triés à partir des `playableTurns`, et sont les coups pour lesquels notre plus grosse composante connexe augmente de plus de 1 lorsqu'on gagne la cellule : Ils sont ceux pour lesquels on rejoint deux territoires.

L'algorithme calcule d'abord les meilleurs coups et les coups jouables. Ensuite, en utilisant le tableau de probabilités codé dans `proba.h`, on choisit le meilleur coup ayant la plus grosse possibilité de gagner, et s'il n'y en a pas, le coup jouable ayant la plus grande probabilité de gagner. S'il n'y a aucun coup de disponible, on passe notre tour.

B - Les possibilités d'évolution

Pour l'évolution de l'IA, nous avons pensé, au lieu de se demander seulement si le coup suivant permet de gagner une composante connexe, de faire un arbre avec cette méthode pour tendre le plus possible vers nos composantes connexes sur plusieurs tours.

Nous avons également pensé à pondérer la probabilité de gagner qui est à ce jour la décideuse finale du coup par d'autres facteurs, comme par exemple le nombre de voisins de la cellule à attaquer : Une cellule avec peu de voisins composant un avantage stratégique.

Une dernière possibilité envisagée est celle de construire un "mur" de défense avec le plus de dés possibles, et de n'avancer que ce dernier petit à petit, pour être certain d'être protégé par un front avec un nombre le plus élevé possible de dés.

7 - Vérification de la mémoire avec Valgrind

Étant donné que le jeu est censé pouvoir se lancer plusieurs fois d'affilée, et qu'à l'intérieur, des structures lourdes sont manipulées régulièrement (par exemple, la map est copiée à chaque coup d'une IA), il nous a semblé important de porter une attention particulière à la mémoire. C'est pourquoi nous avons décidé d'utiliser valgrind pour la vérifier du mieux possible.

Nous nous sommes rapidement aperçus que de grosses fuites mémoires apparaissaient à cause de la librairie SDL2, ce que nous ne pouvions pas gérer nous-mêmes. Ainsi, tous nos tests mémoire se sont fait sans l'affichage.

Nous avons également remarqué une fuite mémoire de 32 bits due aux librairies dynamiques malgré des free de notre part.

Nous avons ajouté la contrainte qu'avant de merger une branche, chacun devait s'assurer que son programme ne présente pas de fuite mémoire.

8 - Débuggage avec GDB

Ce projet a également été une occasion pour nous de découvrir GDB pour le débbugage. En effet, les IDE que nous utilisions ne permettaient pas de regarder par exemple ce qui se situait à l'intérieur des pointeurs, ce qui est possible avec GDB. Notre utilisation de ce dernier n'a cependant pas été très poussée faute de connaissances.