

Rapport de projet

Ce projet a pour but de créer une librairie de classes et de méthodes permettant la gestion de graphes. Celui-ci nous a été axé sur la gestion de pointeurs et ce qui en découle (gestion mémoire, accesseurs, etc.). Bien qu'il soit guidé, nous avons tout de même dû effectuer quelques choix dans la conception de notre librairie que nous expliciterons dans ce compte-rendu.

Sommaire :

- 1- Le diagramme des classes
- 2- L'organisation du développement
- 3- Les choix opérés
- 4- Aide à la compréhension globale
- 5- Fonctionnement des méthodes les plus importantes
- 6- Conclusion

1- Le diagramme des classes

Le diagramme de classes ci-dessous est le résultat de différents ajustements à mesure que le projet a avancé. Par exemple, la structure du parseur a évolué pour passer d'une fonction statique à un couple méthodes + contrôleur/constructeur de graphe. Grâce à la structure de programme proposée, nous avons pu ne pas perdre de temps dans l'élaboration de notre premier diagramme des classes. Néanmoins des adaptations ont été faites, et voici le résultat final :

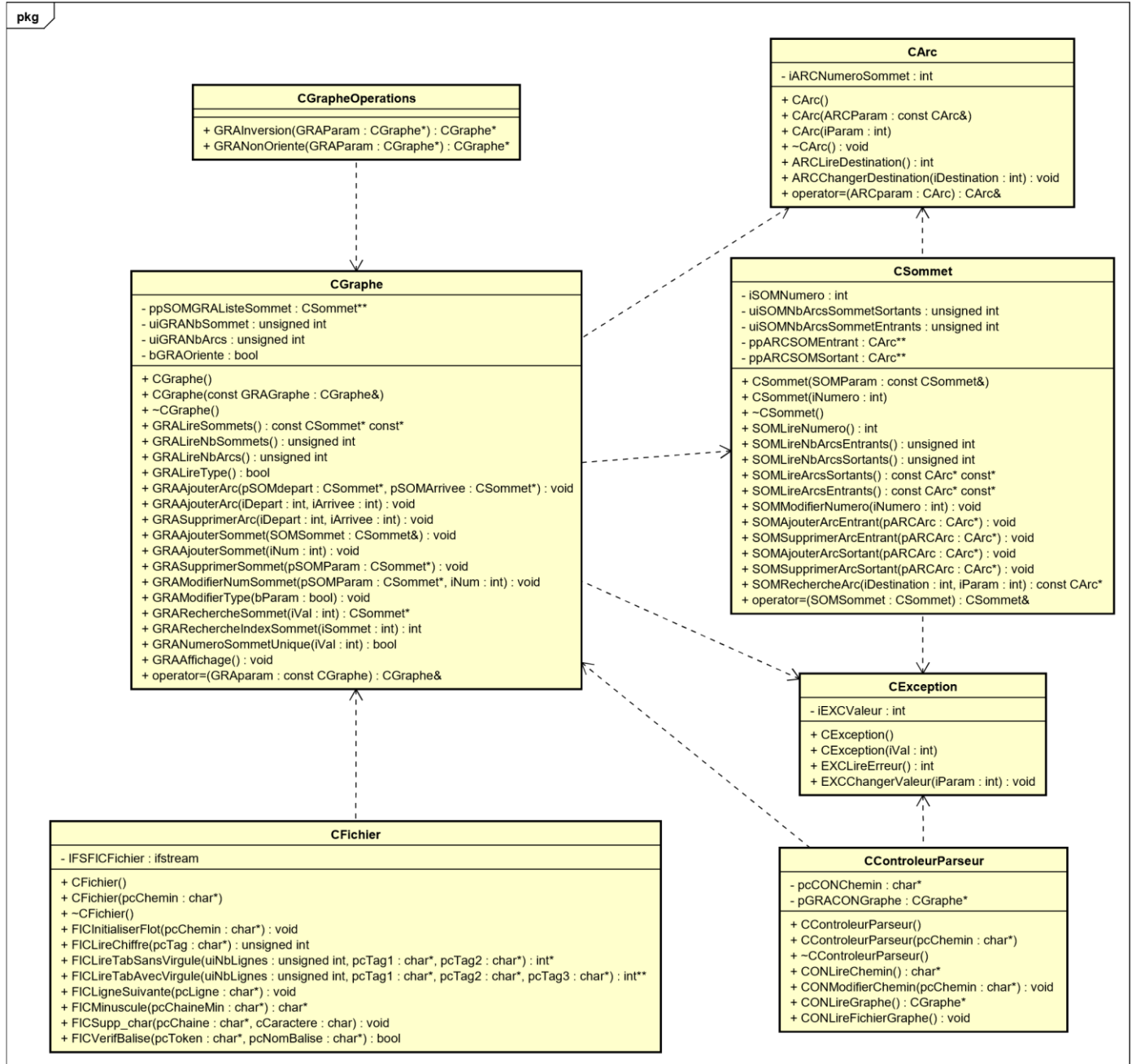


Figure 1 - Le Diagramme des classes

2- L'organisation du développement

Après avoir décidé du diagramme des classes, nous avons commencé le codage du projet.

Nous avons développé dans l'IDE Visual Studio 2022 sur Windows l'ensemble du code de ce projet. Cela nous a permis de bénéficier de beaucoup d'outils de débogage, d'une autocomplétion performante ainsi que de bénéficier du catalogue d'extensions de Visual Studio.

Entre autres, l'utilisation de cet IDE nous a facilité la tâche pour le travail collaboratif : en effet, la version 2022 de Visual Studio intègre nativement une extension GitHub avec tous les outils d'embranchements et de fusion nécessaires au développement parallèle. Ainsi nous avons pu

développer, tester et soumettre de nouvelles fonctionnalités sans compromettre le développement de l'autre.

Néanmoins, lorsque nous devons opérer un développement simultané sur un même fichier sans avoir à attendre les push de chacune des modifications de l'autres, nous avons décidé de profiter de la fonctionnalité Live Share de Visual Studio ce qui nous a permis de gagner beaucoup de temps sur notre développement et évité de nombreux problèmes de fusion de branches.

3- Les choix opérés

Lors du développement du projet nous avons dû faire des choix car aucune solution ne peut être parfaite et des compromis sont par conséquent à faire.

A- Le parseur

Les premiers choix à faire ont été au niveau de la flexibilité du parseur, ainsi qu'à sa structure :

Doit-il accepter que l'utilisateur ne respecte pas la mise en forme spécifiée de manière exacte ? À quel point faut-il sécuriser le parseur avec des exceptions etc... Nous avons donc choisi de sécuriser un maximum le parseur, et surtout à permettre une certaine flexibilité vis-à-vis de la mise en forme : Si des espaces et des tabulations existent dans le fichier, ils sont supprimés et si des lignes sont vides, alors elles sont ignorées.

Cependant, si des valeurs manquent/sont non cohérentes, si le format de fichier est manifestement mauvais ou alors si le nombre de sommets/arcs spécifiés est supérieur au nombre de lignes décrivant ces arcs/sommets, une exception est levée et remontée jusqu'au contrôleur de parseur puis au main.

Comme dit précédemment, nous avons fait un choix sur l'organisation de la lecture du fichier texte passé en argument : Nous avons divisé cette tâche en 2 classes : Une classe CFichier avec un objet de flot en attribut comporte toutes les méthodes nécessaires pour extraire du fichier les types de données nécessaires à l'instanciation d'un graphe et une classe CContrôleurParseur avec le chemin de fichier en attribut qui crée cet objet parseur puis appelle une à une les fonctions de CFichier et instancie un nouveau graphe récupérable dans le main.

B- La gestion des pointeurs

La gestion des pointeurs dans ce projet et particulièrement de la réallocation et suppression de ceux-ci a été une difficulté majeure. Nous avons réalisé au fur et à mesure du projet que le combo new et delete avec realloc n'était pas fonctionnel et avons donc préféré nous passer de la fonction realloc et par conséquent de copier, allouer et détruire nos tableaux de pointeurs manuellement. Cela occasionne une plus grande verbosité du code et plus d'opérations réalisées mais le résultat final est le même et est fonctionnel.

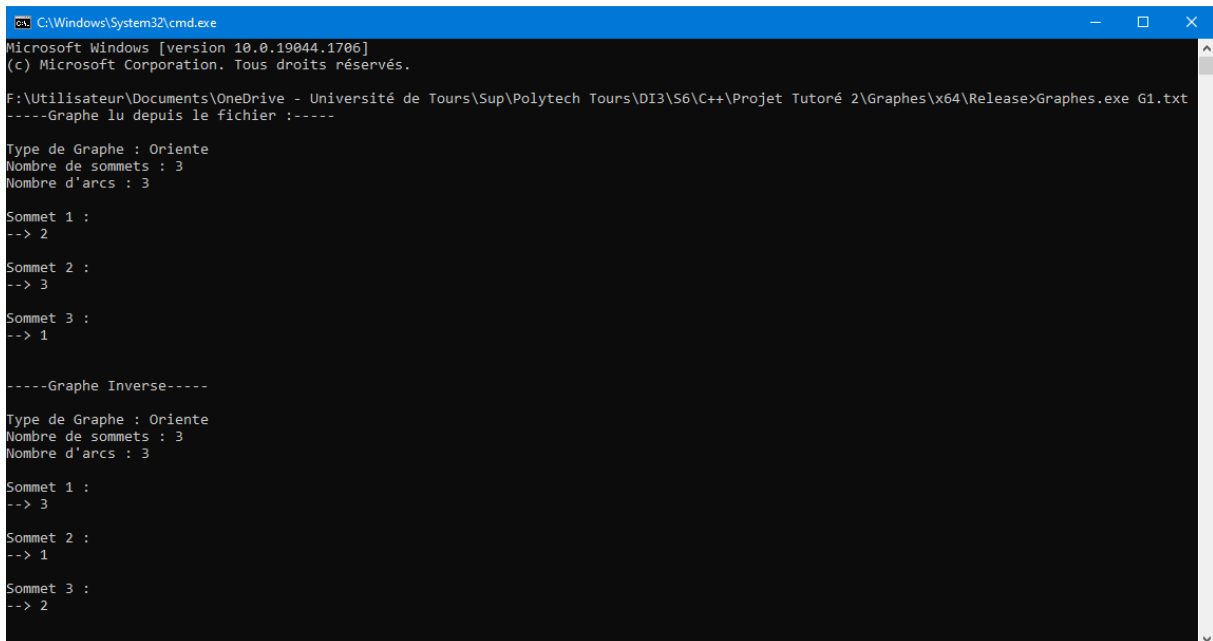
C- La gestion d'un graphe

Nous avons choisi de développer des méthodes dans CArc et CSommet pour permettre une manipulation simple de ces objets. Néanmoins, il n'est pas intéressant, sécurisé et performant d'utiliser directement ces méthodes. C'est pour cela que nous avons intégré des méthodes gérant directement les arcs et les sommets du graphe dans la classe CGraphe. Ce qui veut dire que l'ajout, la suppression et la modification des différentes composantes du graphe est directement gérée par la classe CGraphe.

Cela permet de bénéficier d'une gestion propre de la mémoire, de vérifier l'unicité d'un numéro de sommet, et tout simplement de réaliser toutes les manipulations de graphes nécessaires.

4- Aide à la compréhension globale

Pour utiliser ce programme, l'utilisateur doit passer par l'invité de commande (cmd). Pour ce faire, il doit se placer dans le dossier contenant le fichier "Graphes.exe" et taper "cmd" dans le bandeau. Dans l'invité de commande, il suffit de lancer "Graphes.exe" suivit en arguments du graphe à exploiter.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [version 10.0.19044.1706]
(c) Microsoft Corporation. Tous droits réservés.

F:\Utilisateur\Documents\OneDrive - Université de Tours\Sup\Polytech Tours\DI3\S6\C++\Projet Tutoré 2\Graphes\x64\Release>Graphes.exe G1.txt
-----Graphe lu depuis le fichier :-----

Type de Graphe : Oriente
Nombre de sommets : 3
Nombre d'arcs : 3

Sommet 1 :
--> 2

Sommet 2 :
--> 3

Sommet 3 :
--> 1

-----Graphe Inverse-----

Type de Graphe : Oriente
Nombre de sommets : 3
Nombre d'arcs : 3

Sommet 1 :
--> 3

Sommet 2 :
--> 1

Sommet 3 :
--> 2
```

Figure 2 - Fenêtre de lancement de Graphes.exe

Lorsque l'utilisateur lance le programme, une suite d'opérations se succèdent dans cet ordre :

- Lecture du fichier texte passé en argument et affectation à un CGraphe du résultat.
- Affichage de ce graphe.
- On crée un nouveau graphe issu du premier mais avec une inversion de tous ses arcs.
- Affichage du nouveau graphe inversé.

5- Fonctionnement des méthodes les plus importantes

A. La fonction principale du parseur : `void CONLireFichierGraphe()`

Cette fonction a pour but de générer un objet CGraphe par l'appel des différentes méthodes de la classe CFichier. Nous détaillerons donc ici aussi tout le détail de ces différents appels :

Dans tous les traitements qui existent dans la classe CFichier, et pour nous faciliter les choses, nous faisons appel à des méthodes développées par nos soins du type :

```
char *FICMinuscule(char *pcChaine)
void FICSupp_char(char *pcChaine, char cCaractere)
void FICLigneSuivante (char *pLigne)
```

Elles nous permettent d'utiliser les fonctions de flot tout en ajoutant des couches d'adaptabilité et de sécurité. En effet, lorsque nous passons à la ligne suivante, nous voulons être sur d'obtenir un ligne non vide, sans espaces et sans tabulations. Aussi, nous voulons qu'il soit possible de comparer le nom voulu d'un balise avec ce que l'on trouve dans le fichier sans se soucier de la casse.

Dans l'ensemble de méthodes que forme le parseur, nous sommes amenés à beaucoup utiliser deux fonctions de la bibliothèque standard : strtok et getline. Ces fonctions nous permettent respectivement de découper une chaîne de caractères selon un séparateur et de récupérer la ligne suivante d'un fichier chargé dans un flot ifstream.

Tout d'abord pour le début du passage, on cherche à savoir le nombre d'arc et de sommets qui vont être présents dans le graphe, et donc le nombre de ces valeurs à parser par la suite. Pour cela, après avoir initialisé un objet de type CFichier avec le chemin du fichier texte à lire, on utilise la méthode :

```
unsigned int FICLireChiffre(char *pcChaine)
```

Cette méthode va rechercher dans le fichier une correspondance avec une balise passée en paramètre et renvoyer le chiffre associé à cette balise. En passant en paramètre « NBArcs » et « NBSommets » on récupère les quantités d'arcs et de sommets associés. Ces valeurs sont récupérées dans la fonction principale de CContrôleurParseur et sont ensuite utilisées pour appeler :

```
unsigned int* FICLireTabSansVirgule(const unsigned int uiNbLignes, char *pcTag1, char *pcTag2)
```

```
unsigned int** FICLireTabAvecVirgule(const unsigned int uiNbLignes, char *pcTag1, char *pcTag2, char *pcTag3)
```

Où l'on passe en paramètre le nombre de lignes à récupérer (Nombre d'arcs/de sommets) et le nom de la balise principale ainsi que ses balises secondaires comme le montre l'exemple ci-dessous :

```
puiSommets = FICParseur.FICLireTabSansVirgule(uiNbSommets, (char*)"sommets", (char*)"numero");  
ppuiArcs = FICParseur.FICLireTabAvecVirgule(uiNbSommets, (char*)"arcs", (char*)"debut", (char*)"fin");
```

Figure 3 - Appels de méthodes de CFichier

Ainsi nous récupérerons toutes les informations nécessaires à la génération d'un graphe et ce graphe alloué par new devient attribut de l'objet CContrôleurParseur pour pouvoir être exploité par copie dans le main.

Lors de la lecture du fichier, nombre d'exceptions de mauvaise mise en forme peuvent être levées, dues à une absence de valeurs, une absence de balise, un mauvais nommage de ces balises ou même un format de données incorrectes. Ces exceptions sont récupérées dans CONLireFichierGraphe(), on affiche le message d'erreur et une exception d'arrêt de programme est envoyée dans le main pour stopper l'exécution du programme.

B. Les méthodes d'ajout/suppression de sommet

```
void GRAAjouterSommet(int iNum)
```

```
void GRAAjouterSommet(CSommet& SOMSommet)
```

```
void GRASupprimerSommet(CSommet *pSOMSommet)
```

Ces méthodes sont celles qui effectuent le plus d'opérations parmi toutes les méthodes de CGraphe. En effet, l'ajout d'un sommet implique de l'ajouter dans le tableau de sommets au sein de la classe de graphe, d'augmenter le compteur de sommets dans le graphe et surtout s'assurer que le numéro de

sommet est unique. En effet, vu que la structure du projet impose qu'un arc entre 2 sommets stocke un entier pour identifier le sommet vers lequel il est dirigé et pas directement un pointeur vers le sommet en question, il est absolument nécessaire que chaque numéro de sommet soit identique. Pour cela on fait appel à une méthode de CGraphe qui renvoie un booléen pour savoir si le numéro du sommet à ajouter n'est pas déjà pris :

```
bool GRANumeroSommetUnique(int iVal)
```

Si cette condition n'est pas remplie, alors on demande directement à l'utilisateur un numéro dans la console et on s'assure que ce qui est entré est bien au format d'entier, sinon, on reformule la demande :

```
while (!GRANumeroSommetUnique(SOMSommet.SOMLireNumero())) {  
    cout << "Les numeros de sommets utilises sont : " << endl;  
    for (uiBoucle = 0; uiBoucle < GRALireNbSommet(); uiBoucle++) {  
        cout << GRALireSommets()[uiBoucle]->SOMLireNumero() << " ";  
    }  
    cout << endl;  
    cin >> pcEntree;  
    iNum = atoi(pcEntree);  
    SOMSommet.SOMModifierNumero(iNum);  
}
```

Figure 4 - Demande à l'utilisateur d'un numéro de sommet unique

On part du postulat que l'objet CSommet à ajouter n'a pas d'arcs et qu'ils sont à ajouter ultérieurement.

Pour la suppression d'un sommet, on doit s'assurer de la bonne désallocation de l'objet dans le graphe mais aussi de la déconnection du sommet avec ceux avec qui il avait des arcs. Ainsi on est amené à parcourir le tableau des arcs entrants et sortants du sommet à supprimer puis de veiller à bien supprimer les arcs en question des tableaux d'arcs des sommets pointés. Ce travail passe par une recherche des pointeurs des sommets en fonction des entiers contenus dans les graphes. Puis de la bonne suppression du sommet dans le graphe.

Conclusion

Ce projet nous a permis de mettre en application tout ce que nous avons pu voir en théorie (la modélisation orientée objet avec le diagramme des classes, l'algorithmique orientée objet et bien sûr le C++) et de capitaliser sur l'expérience acquise lors de la partie 1 de ce projet. Nous avons également pu faire preuve d'encore plus d'autonomie que la partie 1 qui elle-même était bien plus importante que les précédents projets. Nous avons continué à développer notre vision globale d'un projet et à prendre du recul par rapport à notre travail : en effet, nous avons continué à chercher et limiter les conditions anormales mais prévisibles de fonctionnement et gérer les exceptions en conséquence. L'ensemble de ce que nous avons appris ici nous sera encore très utile par la suite.