

GraphY

Rapport

Projet de semestre

Browne Champion Djomo Hardy Richoz Rochat

Resp. René Rentsch

HEIG-VD

30 mai 2016

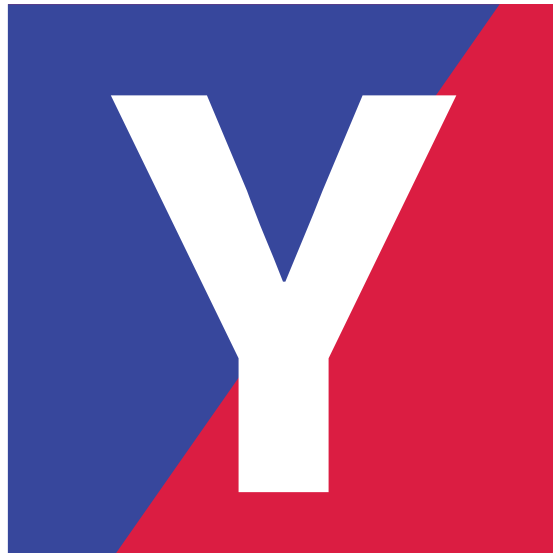


Table des matières

1	Introduction	5
1.1	Objectif	5
1.2	Conception générale	5
2	Cadre de développement	6
3	Interface	6
3.1	Fenêtre principale	6
3.1.1	Conception générale	6
3.1.2	Console	6
3.2	Auto complétion	7
3.2.1	Implémentation du dictionnaire	7
3.2.2	Auto-complétion dans la GUI	7
3.3	Aide utilisateur	7
3.3.1	Conception générale	7
4	Représentation graphique des graphes	7
4.1	Widget de visualisation	7
4.1.1	Fonctionnement de Qt	7
4.1.2	Diagramme de classe	9
4.1.3	Drag & drop des sommets	10
4.1.4	Ajustement des arcs/arêtes	10
4.1.5	Notion de "Bounding rect"	10
4.2	Fabriques des éléments graphiques	10
4.3	Widget	12
4.4	Exportation	12
4.5	Réutilisation	13
5	Interpréteur	13
5.1	Normalisation du langage	13
5.1.1	Analyse des types et des opérations	13
5.2	Architecture	16
5.2.1	Interface utilisateur	16
5.2.2	Diagramme de classes	17
5.2.3	Répertoires	18
5.3	Exemple	18
5.3.1	Fonctions incluses	19
6	Graphes	20
6.1	Préface	20
6.2	Architecture	21
6.2.1	Diagramme de classe	21
6.2.2	La classe "Vertex"	21
6.2.3	La classe "Edge"	22
6.2.4	Classe "Graph"	23
6.2.5	Structure de donnée de la classe "Graph"	25
6.2.6	Détails d'implémentation de la structure	27
6.2.7	Création de graphes	28
6.3	Algorithmes	29
6.3.1	Pattern Visitor	29
6.3.2	Liste des algorithmes implémentés	31
6.3.3	Utilisation	31
7	Conclusion	32
7.1	Application fournie	32
7.2	Planification	33
7.3	Travail de groupe	34

8	Annexes EGLI	36
8.1	Grammaire EBNF	36
8.1.1	Modifications	37
8.2	Flux des données	38
8.3	Implémentation du parseur	39
8.4	Mémoire virtuelle	39
8.4.1	Table des variables	39
8.4.2	Table des temporaires	40
8.4.3	Tableau dynamique hétérogène	41
8.4.4	Number, Edge et Vertex	42
8.5	Table des fonctions	44
8.5.1	Interfaçage	45
8.5.2	Surcharge	47
8.6	Règles de transformation	48
8.6.1	Modifications	49
8.6.2	Fonctions internes	50
8.6.3	Arbre d'appels des traitements	50
8.6.4	Vérification des types	51
9	Annexes projet	51
9.1	Cahier des charges	51
9.2	Journaux de travail	55
9.3	Planification initiale	67

Remerciements

Nous tenons tout d'abord à remercier M. Prof. René Rentsch pour son suivi et ses conseils durant toute la période du travail. Nous voulons aussi remercier M. Prof. Claude Evéquoz pour son aide à la vérification de la grammaire de notre langage. Enfin, merci également à M. Prof. Jean-François Hêche pour ses conseils sur l'implémentation des algorithmes de graphes.

1 Introduction

Dans le cadre du cours PRO en deuxième année du cursus de Bachelor de la HEIG-VD, il a été demandé de réaliser un projet de semestre sur une période de 10 semaines. Après l'acceptation du cahier des charges à la semaine 4, le projet a pu débuter. Celui-ci porte sur une application permettant de traiter des graphes, de lancer des algorithmes dessus, de les dessiner et enfin de pouvoir les exporter en format d'image vectoriel.

1.1 Objectif

Les objectifs principaux du rapport peuvent être décomposés en quatre parties. Premièrement, il doit être possible de saisir un graphe et qu'il soit stocké dans l'application. Ensuite, des algorithmes classiques des graphes (Dijkstra, parcours en profondeur/largeur, etc...) doivent pouvoir être appliqués au graphe précédemment saisi. Puis, il doit pouvoir être dessiné et remanié avec le curseur pour permettre de l'exporter en image vectorielle. Enfin, toutes ces parties doivent être coordonnées au moyen d'un langage spécifique à l'application. Toutes les spécifications précises sont disponibles dans le cahier des charges fourni en annexe.

1.2 Conception générale

La conception générale de l'application a été faite en couches pour deux raisons. Premièrement, cette décomposition permet de répartir le travail plus facilement et de manière plus efficace entre les personnes. En effet, cela réduit les dépendances au niveau du développement et donc le temps passé à attendre sur le travail d'un autre, ainsi que le nombre de conflits sur le gestionnaire de versions. Deuxièmement, cette approche permet d'avoir des composants indépendants (voir **figure 1**) et donc réutilisables, puisque les couches inférieures n'ont pas connaissance des couches supérieures et proposent des interfaces indépendantes.

La première couche à mentionner, celle avec laquelle l'utilisateur interagit est l'interface graphique (GUI). Celle-ci propose une console pour saisir des commandes, ainsi que quelques menus. Elle est également liée à l'aide utilisateur, qui comporte une fenêtre de navigation HTML. La GUI est interfacée avec l'interpréteur qui vérifie la validité des commandes tapées et renvoie un message d'erreur en cas de mauvaise saisie. Si, au contraire, la commande est correcte, l'interpréteur la transmet soit à la couche graphe, soit à la couche algorithme. En parallèle à cet interpréteur, la couche de dessin accède également à la couche graphe pour en dessiner des représentations.

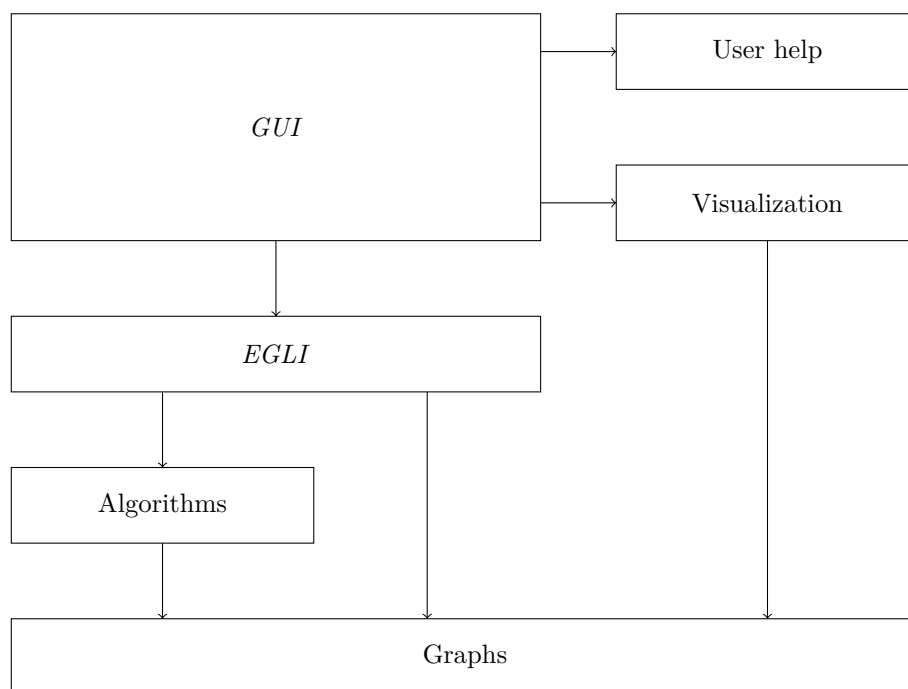


FIGURE 1 – Couches logicielles

Ce rapport est organisé de manière à partir des couches hautes (*GUI*) pour descendre petit à petit dans les couches basses.

2 Cadre de développement

L'application est développée à l'aide du langage C++ (standard C++11) et de la bibliothèque Qt (version 5.5.1). Elle est compilée avec MinGW 4.9.2 32 bits sur Windows. La version de la librairie Boost utilisée pour le parseur est la 1.60.0. Afin que le code source soit écrit dans le même style par tous les membres du groupe, nous avons décidé d'utiliser le Coding Style de Qt [1] qui correspond bien à nos besoins. Nous avons également utilisé GitHub pour gérer la mise en commun du code.

3 Interface

3.1 Fenêtre principale

3.1.1 Conception générale

Afin de mettre en place un principe de session de travail, un système d'onglets multiples est proposé, chacun contenant une console, afin de pouvoir travailler en parallèle sur plusieurs graphes.

3.1.2 Console

Qt ne proposant pas directement de Widget faisant office de console, il a fallu en créer un adapté au besoin de l'application. La base de la console est un QTextEdit, un composant permettant la saisie libre de texte. Le QTextEdit autorisant justement de saisir du texte où bon nous semble, il a fallu mettre en place des fonctionnalités limitant l'utilisateur dans les endroits où il peut saisir du texte. Dans un premier temps, un buffer contenant le texte saisi par l'utilisateur a été implémenté.

Ensuite, pour synchroniser les données réellement saisies et l'affichage, un curseur géré manuellement a été mis en place. En effet le curseur de base est sensible au clic de la souris, ce qui lui permettrait de pouvoir insérer du texte n'importe où. Grâce au curseur créé, on s'assure que le texte sera inséré au bon endroit dans l'affichage. Le menu contextuel par défaut aussi posait problème, car il implémentait des fonctionnalités allant à l'encontre de l'intégrité de la console, comme par exemple la fonction "couper" ou encore "supprimer" qui permettait d'enlever du texte de la console. Un nouveau menu, proposant uniquement des fonctionnalités ne supprimant pas l'affichage actuel, a été créé. Ainsi, grâce à celui-ci, il est possible de "Copier", "Coller", et "Sélectionner tout" du texte d'une console. La fonction "Coller" insère le texte à partir du curseur géré manuellement pour empêcher des incohérences dans l'application.

3.2 Auto complétion

3.2.1 Implémentation du dictionnaire

La première étape pour permettre l'implémentation de l'auto complétion est le choix d'une structure de données adéquate. Après quelques recherches, le *ternary search tries* se montre le plus adéquat. En effet, sa mise en œuvre permet une recherche très rapide. Pour une recherche réussie (*search hit*), sa complexité est de l'ordre de $O(L + \ln(N))$ avec L la longueur de la chaîne recherchée et N le nombre de mots dans le dictionnaire [6]. Une recherche ratée (*search miss*) prend quant à elle $O(\ln(N))$. Il est vrai qu'une hash map effectue toutes ses opérations en moyenne en $O(L)$, mais le temps de hachage et non-négligeable et la performance globale en temps et en espace mémoire est souvent moins bonne [7].

3.2.2 Auto-complétion dans la GUI

Au final, l'auto-complétion proposée par Qt a pu être adaptée à l'utilisation dans la console principale, ce qui a rendu les efforts consacrés à l'implémentation d'un dictionnaire efficace inutiles. Le comportement de l'auto-complétion n'est pas exactement celui initialement prévu, mais son utilisation a permis un gain de temps global.

3.3 Aide utilisateur

3.3.1 Conception générale

Les fichiers d'aide utilisateur ont été écrits au format HTML, afin qu'ils soient rapidement rédigés et que l'utilisateur puisse naviguer d'une page à l'autre. L'interface de l'aide contient également une barre de recherche qui utilise un fichier de mots-clés pour effectuer ces recherches. Ceci a été fait, plutôt que de chercher le texte dans les fichiers, afin qu'une recherche avec un mot commun du langage, tel que *graph*, *edge* ou *vertex* n'affiche pas toutes les pages de l'aide, puisque ces mots se trouvent dans la majorité des fichiers.

4 Représentation graphique des graphes

4.1 Widget de visualisation

4.1.1 Fonctionnement de Qt

Qt possède un système de dessin complexe, le *Graphics View Framework*. Cet outil est basé sur le principe suivant : trois types d'éléments se partagent différentes responsabilités.

La scène (classe `QGraphicsScene`) :

- Fournit une interface afin de gérer les éléments graphiques.
- Gère les états des éléments graphiques.
- Propage les événements aux éléments graphiques.

La scène est donc un conteneur pour tout ce qui sera visible par l'utilisateur. Qt fournit des méthodes pour ajouter, supprimer des éléments, mais aussi pour trouver un élément à une certaine coordonnée. Lors de la sélection d'un élément, c'est également la scène qui s'occupe de l'informer d'une action.

La vue (classe `QGraphicsView`) :

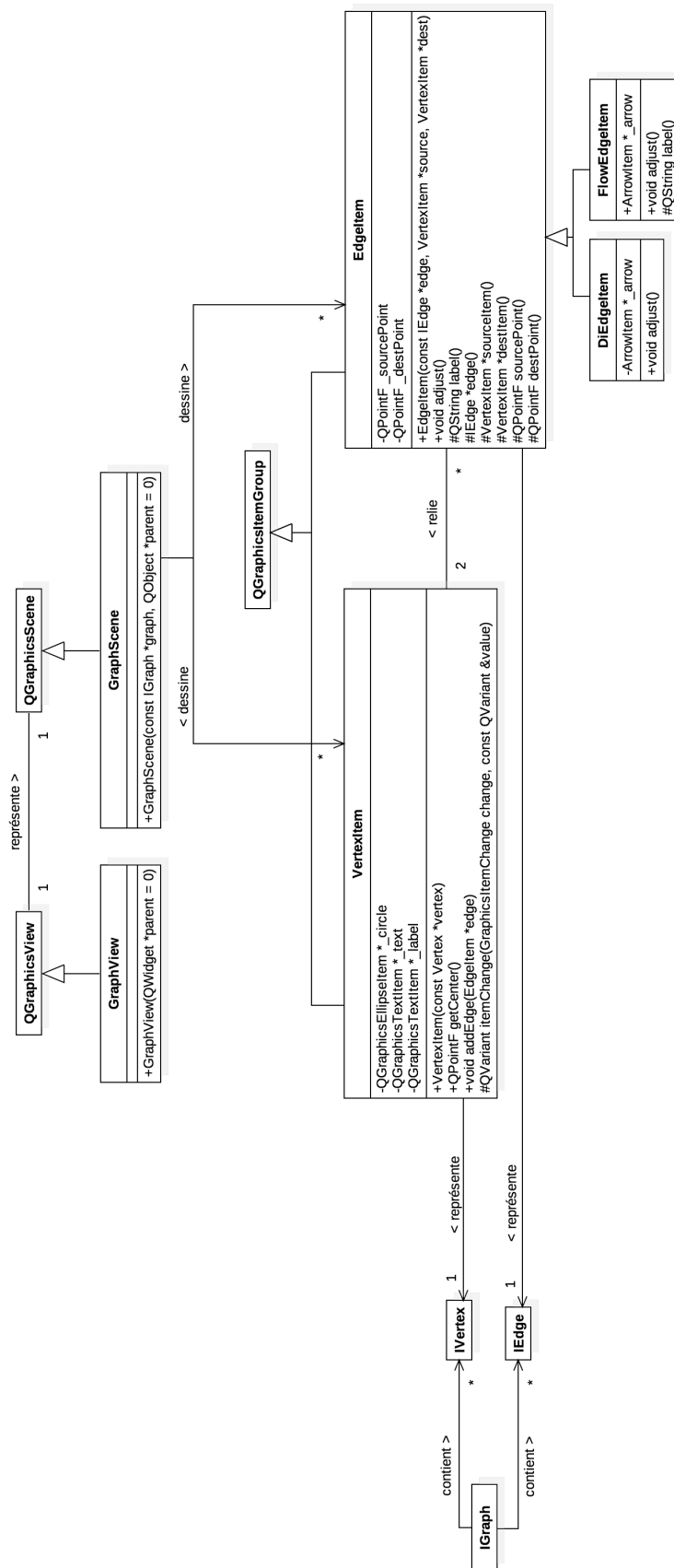
La vue se charge d'afficher le contenu d'une scène (plusieurs vues peuvent très bien afficher la même scène). Ici, la scène peut être perçue comme un modèle du design MVC.

Elle s'occupe d'afficher tout ou une partie de la scène, c'est à dire que si cette dernière est grande, trop grande pour la fenêtre ou pour la zone à disposition, la vue va s'occuper d'ajouter des barres de défilement. Ceci engendre que les coordonnées peuvent différer entre une vue et sa scène. La vue recevant en premier les actions de l'utilisateur, va les transmettre à la scène en prenant soin d'adapter les coordonnées si besoin. Par exemple, un clic en haut à gauche de la fenêtre sera en coordonnées (0,0) pour la vue, mais si l'utilisateur a auparavant scrollé horizontalement et/ou verticalement, la coordonnée réelle du clic sur la scène pourrait être (0,100), (130, 10), etc. Cette adaptation est donc extrêmement importante. Des méthodes sont également mises à disposition par Qt afin de manuellement demander la correspondance d'une coordonnée vue -> scène ou scène -> vue.

Les éléments graphiques (classe `QGraphicsItem` et sous-classes) :

Ces classes représentent des éléments graphiques de base, une ellipse (`QGraphicsEllipseItem`), un rectangle (`QGraphicsRectItem`), un texte (`QGraphicsTextItem`), etc. Il est possible de les paramétrer en leur fournissant un objet `QPen` personnalisé (pour le contour) et/ou un objet `QBrush` (pour le fond) et en utilisant leur

interface publique respective. Il est également possible de spécialiser une classe de Qt afin de créer ses propres éléments. Ces objets possèdent une fois de plus leur propre système de coordonnées. La position qu'il leur est attribuée par la scène devient leur point d'origine et là aussi des méthodes permettent des conversions.



Afin de respecter la philosophie de Qt, j'ai créé des classes spécialisées pour chaque partie. La classe **GraphView** n'a pas forcément d'utilité pour l'instant, mais pour l'éventuel avenir du projet, celle-ci permettra de gérer des événements plus précis (actuellement nous n'avons implémenté que le drag and drop des sommets et ceci est automatiquement géré par Qt).

La classe **GraphScene** prend en paramètre un pointeur de IGraph. C'est cet objet qui va s'occuper de créer tous les éléments graphiques nécessaires à sa représentation.

Puis les éléments graphiques **VertexItem** et **EdgeItem** (ainsi que ses sous-classes **DiEdgeItem** et **FlowEdgeItem**) permettent respectivement de dessiner les sommets et les arcs/arêtes. Ces classes ne sont pas directement de type QGraphicsItem, mais de QGraphicsItemGroup qui est en fait un conteneur de plusieurs QGraphicsItem. En effet, un sommet est composé d'un cercle (réellement d'une ellipse) d'un texte pour l'identifiant affiché et d'un autre texte pour son label. Les arêtes (EdgeItem) qui sont des simples lignes avec un label, les arcs (DiEdge) qui ajoutent une flèche pour la direction et les flux (FlowEdge) qui ajoutent également une flèche, mais aussi les capacités. Chacun des éléments graphiques ajoute et gère tous les autres éléments graphiques dont il a besoin.

Finalement, la scène se charge de donner une position aux sommets. Pour ce projet, le but était de placer les éléments simplement sur une grille et de laisser la possibilité aux utilisateurs de modifier cette disposition. La scène calcule donc une coordonnée basée sur une grille de quatre pour chaque sommet. Les arcs/arêtes eux, se placent automatiquement en fonction des sommets qu'ils relient.

La classe Graphe met à disposition des identifiants uniques croissants. Ceci m'a permis de pouvoir faire le lien entre un sommet et un sommet graphique afin de savoir, d'après un IEdge, quels sommets graphiques relier.

4.1.3 Drag & drop des sommets

La classe **VertexItem** représentant un sommet, utilise une fonctionnalité de Qt qui permet d'automatiquement implémenter le drag & drop, au travers de méthodes appelées dans le constructeur de la classe. Tout est donc automatique.

J'ai néanmoins dû faire en sorte que si un sommet est déplacé, les arcs/arêtes liées s'adaptent. Qt met à disposition une méthode **itemChange()** sur les éléments graphiques afin d'être averti lors d'événements. Surcharger cette méthode a permis d'effectuer une action si le type d'événement est un drag & drop, à savoir ajuster tous les arcs/arêtes liés.

4.1.4 Ajustement des arcs/arêtes

Les objets de type **EdgeItem** possèdent une méthode **adjust()** lui permettant de s'ajuster aux sommets source et destination. Cette méthode est donc appelée à la construction de l'élément et lorsque qu'un sommet lié a été déplacé (comme expliqué précédemment). Les lignes ne reliant pas le centre des cercles représentant les sommets, mais s'arrêtant à l'extérieur de ces derniers, il a fallu effectuer quelques calculs à l'aide du théorème de Thalès afin de déplacer les points dans le plan.

4.1.5 Notion de "Bounding rect"

La notion de **Bounding rect** est importante lors du dessin des différents éléments. Il s'agit d'une zone indiquant à la vue qu'est-ce qui doit être dessiné et effacé. Elle doit être la plus précise possible afin de ne pas demander au programme de redessiner quelque chose qui n'a pas changé (un arrière-plan par exemple), mais elle doit également englober totalement l'élément. C'est par exemple important lors du déplacement d'un élément graphique. La vue va redessiner ce qui se trouvait dans la zone précédente et dans la nouvelle zone, afin de faire disparaître l'élément de son ancien emplacement et de le dessiner dans son nouvel emplacement. Si la zone ne couvre pas tout l'élément, des parties de celui-ci pourraient ne pas être mises à jour et laisser des résidus visuels.

4.2 Fabriques des éléments graphiques

Revenons sur la classe **GraphScene**. Celle-ci peut recevoir en paramètres tout types de graphes (normaux, dirigés et de flot). Elle doit être capable de créer les bons éléments graphiques. Pour ce faire, elle essaie de caster dynamiquement le IGraph en Graph, DiGraph et FlowGraph ce qui permet d'en connaître le type réel.

La scène parcourt ensuite chaque sommet, puis chaque noeud. Les sommets ne changent pas en fonction du type de graphe, en revanche les noeuds nécessitent de créer les bons types d'éléments graphiques. J'ai donc décidé d'appliquer le design pattern **Abstract Factory** qui permet de créer une factory d'un certain type, puis d'appeler des méthodes de création qui vont se charger d'instancier directement les bonnes classes.

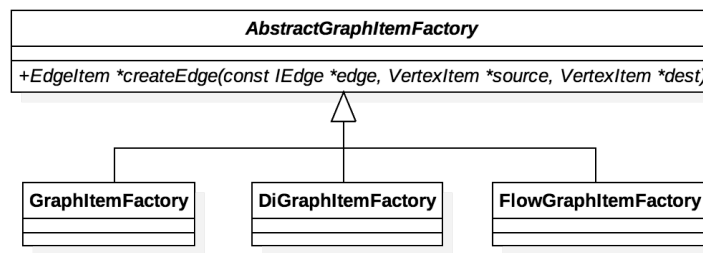


FIGURE 3 – Diagramme de classes de la fabrication des éléments graphiques

Les factories sont donc toutes du même type de base, une classe abstraite qui définit les méthodes de création. Il y a ensuite une classe spécialisée par type de graphe, par famille (normal, dirigé et de flot). Ces sous-classes ont ensuite simplement à retourner une instance de l'élément graphique correspondant à ce type.

Voici un exemple d'utilisation :

```

// Surcharge d'une methode de creation
// pour retourner une instance de la famille
EdgeItem *DiGraphItemFactory::createEdge(const IEdge *edge, VertexItem *source,
VertexItem *dest)
{
    return new DiEdgeItem(edge, source, dest);
}
    
```

```

// Creation de la factory correspondante a la bonne famille
AbstractGraphItemFactory *factory;
if (dynamic_cast<const FlowGraph *>(graph)) {
    factory = new FlowGraphItemFactory();
}
else if (dynamic_cast<const DiGraph *>(graph)) {
    factory = new DiGraphItemFactory();
}
...

// Creation d'un objet EdgeItem sans se soucier de son reel type
EdgeItem *edgeItem = factory->createEdge(
    pointeurDeEdge,
    sommetSource,
    sommetDestination
);
    
```

4.3 Widget

Afin de simplifier l’affichage d’un graphe depuis la partie GUI, j’ai fait en sorte qu’un nouvel objet s’occupe d’instancier les objets nécessaires (vue et scène) et de configurer ces derniers. Il s’agit de la classe **GraphWidget** qui prend en paramètre un pointeur de **IGraph** à traiter. Ce widget peut ensuite être incorporé dans n’importe quelle layout, avec n’importe quelle dimension, sans se soucier du fonctionnement interne.

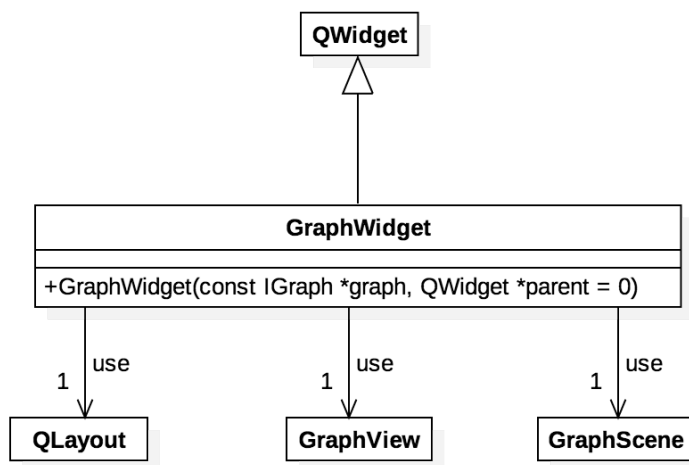


FIGURE 4 – Diagramme de la classe GraphWidget

Cette classe hérite de **QWidget**, une classe de Qt permettant justement d’être ajoutée et affichée dans un layout. Le widget possède également un layout, nécessaire pour y ajouter et afficher la vue du graphe.

4.4 Exportation

L’exportation des graphes (uniquement au format SVG pour ce projet) s’effectue à l’aide de la classe **GraphExporter**. Cette classe met à disposition des méthodes statiques permettant d’exporter au format SVG. Ces méthodes prennent un pointeur de **IGraph** en paramètre, elles génèrent ensuite la scène correspondante au graphe, puis l’enregistre dans un fichier.

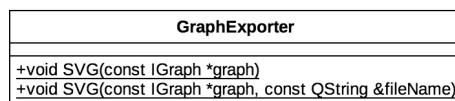


FIGURE 5 – Diagramme de la classe GraphExporter

Pour l’exportation SVG, il y a deux méthodes. Une avec un chemin de fichier en paramètre, l’autre sans. La méthode sans chemin va automatiquement ouvrir une fenêtre invitant l’utilisateur à choisir l’emplacement et le nom du fichier à enregistrer (à l’aide de la classe **QFileDialog** de Qt). Celle-ci va ensuite appeler la deuxième méthode avec le nom du fichier spécifié.

Pour l’exportation, la scène est simplement rendue dans un objet spécifique à la place de celui gérant l’affichage. Il s’agit d’un objet **QSvgGenerator**. Ce dernier prend en paramètre le chemin du fichier et la taille du dessin SVG à créer.

Le générateur s’utilise ensuite simplement en appelant l’une de ses méthodes statiques.

```

GraphExporter::SVG(myGraph);
// ou
GraphExporter::SVG(myGraph, "/path/to/svg/file.svg");
  
```

4.5 Réutilisation

Un problème se posait quant à la génération des graphes. Si un utilisateur affichait un graphe, déplaçait les sommets pour obtenir un affichage plus lisible et qu'il souhaitait ensuite l'exporter, le fichier contiendrait l'affichage par défaut (en grille actuellement) et pas ces modifications. Ceci représente également une perte de temps de régénérer l'affichage entier, alors que celui-ci a déjà été fait.

Nous avons donc décidé d'implémenter un manager de scènes (GraphScene) qui se charge de créer une nouvelle scène à partir d'un IGraph seulement si nécessaire. Ce manager devait donc avoir connaissance des changements survenus sur les graphes. En clair, un graphe qui n'a pas changé (sommets, arêtes, etc.) doit utiliser la même scène d'un affichage à l'autre.

Il n'était pas facile de mémoriser l'état des graphes. Nous avons donc créé une méthode de hachage d'un IGraph, prenant en compte tous les attributs des graphes. Le hash obtenu permet de distinguer un graphe avant et après une modification. Les scènes sont donc stockées par le manager et liées à un hash. Lors de la demande de création d'une scène, les hashes sont comparés et s'il y a une concordance, la scène existante est retournée à la place qu'une nouvelle soit créée.

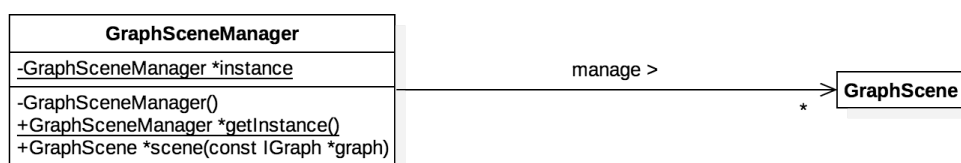


FIGURE 6 – Diagramme de la classe GraphSceneManager

Cela permet donc d'économiser du temps et permet aux utilisateurs de garder en mémoire les modifications qu'ils apportent visuellement aux graphes.

Les objets désirant une instance de la classe GraphScene doivent donc maintenant toujours passer par le manager.

5 Interpréteur

Dans le cadre de l'application, l'utilisateur est amené à entrer des commandes lui permettant de créer et de modifier des graphes, tout autant que d'appeler des fonctions effectuant différents traitements (algorithmes, lecture depuis un fichier, ...).

Note : les détails d'implémentation peuvent être consultés dans l'annexe 8.

5.1 Normalisation du langage

Il est nécessaire de définir une grammaire claire sur la syntaxe des commandes, ainsi que leur sémantique.

5.1.1 Analyse des types et des opérations

Premièrement, il faut définir les types disponibles dans le langage. Pour cela, il est intéressant de partir des algorithmes de graphe et de voir les types de résultats que nous attendons en sortie, ainsi que les types de paramètres dont nous aurons besoin :

- Boolean : un graphe a-t-il un cycle ? est-il eulérien ? ...
- Number : poids d'un arc, indice d'un sommet, ...
 - Integer : pour les index, ...
 - Float : pour les poids, ...
- String : label d'un sommet, nom d'une fonction, ...
- Array : liste d'arêtes, matrices (Floyd-Warshall), ...
- Graph : le graphe à proprement parler
 - Vertex : un sommet, son indice, son poids, ...
 - Edge : une arête/arc, son poids, ...

Puisque nous avons à présent une idée des types disponibles, il faut définir leur domaine ainsi que les opérations disponibles et leur syntaxe. On fait le choix délibéré de se concentrer sur les opérations concernant les graphes, les opérations simples comme additionner deux nombres où les comparaisons ne sont pas prévues. Cependant la base du langage doit permettre de les définir plus tard.

Boolean

Domaine	True ou False	
Opérations	Déclaration	Boolean a = True;
	Affectation	a = False; a = f();
	Lecture	a; f(a); Boolean b = a;

Number

Domaine	Integer et Float
---------	------------------

Integer

Domaine	Entier signé sur 32 bits	
Opérations	Déclaration	Integer a = -20;
	Affectation	a = 2; a = f();
	Lecture	a; f(a); Integer b = a;

Float

Domaine	Nombre à virgule flottante sur 32 bits	
Opérations	Déclaration	Float a = -32.4;
	Affectation	a = 4.0; a = f();
	Lecture	a; f(a); Float b = a;

String

Domaine	Ensemble de zéros ou plusieurs caractères ASCII	
Opérations	Déclaration	String a = "Hello";
	Affectation	a = "World"; a = f();
	Lecture	a; f(a); String b = a;

Array

Domaine	Tableau dynamique hétérogène	
Opérations	Déclaration	Array a = [1.0, "Salut", 3];
	Affectation	a = [4, 5];
	Lecture	a; f(a); Array b = a;
	Accès	Integer c = a[1];

Vertex

Domaine	Sommet avec des informations supplémentaires facultatives	
Opérations	Déclaration	Vertex a = (1); Vertex a2 = (2::3); (id:label:weight:max_capacity:min_capacity)
	Affectation	a = (1:"Yverdon"); a = f();
	Lecture	a; f(a); Vertex b = a;

Edge

Domaine	Arête/arc avec des informations supplémentaires facultatives	
Opérations	Déclaration	Edge a = (1--2); Edge a2 = (2<-3:5); (connection[id]:weight:label:max_capacity:min_capacity) (arête: --, arcs: ->, <-)
	Affectation	a = (1->2:"A1"); a = f();
	Lecture	a; f(a); Vertex b = a;

Graph

Domaine	Ensemble de sommets et d'arêtes/arcs	
Opérations	Déclaration	Graph a = {0, 1, 2:A, 1->2:3:::2}}; (si on ne veut pas écrire tous les sommets : a = {#3, 0->1, 0->2};) (les parenthèses pour les Vertex et Edge ne sont plus nécessaires)

Affectation	<code>a = {0, 1, 0-1}; a = f();</code>
Lecture	<code>a; f(a); Graph b = a;</code>
Ajout/modification	<code>a += (1<-2:4);</code>
Suppression	<code>a -= [(3), (1->2)];</code>

Ce tableau nous donne à présent une vue assez claire de nos besoins, cependant certaines opérations des types complexes (**Array**, **Vertex**, **Edge** et **Graph**) méritent d'être approfondies :

- **Array** :
 - Index de 0 à n-1
 - Accès en dehors des bornes → Exception
- **Vertex** :
 - Seul le premier paramètre (`id[Integer]`) est obligatoire, et il ne doit pas être négatif (`id` est le nom, et entre `[]` c'est son type)
 - Les valeurs par défaut sont `label[String]="", weight[Number]=0, max_capacity[Number]=min_capacity, min_capacity[Number]=0`
- **Edge** :
 - Seul le premier paramètre (`connection`) est obligatoire
 - Le paramètre `id[Integer]` doit être positif ou nul et sa valeur par défaut est 0
 - Les valeurs par défaut sont `weight[Number]=0, label[String]="", max_capacity[Number]=min_capacity, min_capacity[Number]=0`
 - L'`id` est un identifiant "local" à la `connection`, c'est-à-dire par exemple que (0->1) et (1->0) auront tous deux l'`id` à 0, et cela ne pose pas problème
- **Graph** :
 - La création d'un graphe vide est permise (`Graph g = {};`)
 - Le raccourci d'écriture pour le nombre de sommets (`#3`) doit se trouver au début
 - Pour que le **Vertex** d'`id n` soit créé, il faut que tous les `id` de 0 à n-1 existent déjà, sinon → Exception
 - Dans le cas d'arêtes/arcs multiples, par exemple `{#2, 0->1, 0->1}`, le paramètre `id` du second **Edge** est incrémenté → `0->1[0]` et `0->1[1]`
 - Pour qu'un **Edge** soit créé, il est nécessaire que les deux **Vertex** soient déjà créés, sinon → Exception
 - Ajouts / modifications :
 - Les types acceptés en opérande de droite sont **Vertex**, **Edge** ou un **Array** de ces deux types
 - Pour les **Vertex**, si l'`id` existe déjà dans le graphe, c'est une modification, sinon c'est un ajout
 - Pour les **Edge**, si la `connection` existe déjà dans le graphe mais que l'`id` est omis, alors c'est un ajout, sinon c'est une modification (sauf si l'`id` n'existe pas, dans ce cas c'est un ajout)
 - La modification est cumulative, par exemple si on fait `Graph g = {0:"Yverdon"}; g += (0::2);`, alors le **Vertex** résultant est `(0:"Yverdon":2)`
 - Suppressions :
 - Les types acceptés en opérande de droite sont **Vertex**, **Edge** ou un **Array** de ces deux types
 - S'il n'y a rien à supprimer, alors il ne se passe rien (pas d'exception)
 - La suppression d'un **Vertex** entraîne la suppression des **Edge** associés
 - Pour les **Edge**, si l'`id` est omis, alors toutes les `connection` sont supprimées

Notons que les erreurs sont gérées au travers d'exceptions.

Maintenant que les types des variables et leurs opérations de base sont définis, on veut pouvoir effectuer d'autres traitements sur ces variables (ajouter une nouvelle opération ou appliquer un algorithme). Cela va se faire via des fonctions prédéfinies (la définition de fonctions n'est pas prévue).

Prototype d'une fonction : `R f(T1, T2, ...)`; avec `R` le type de retour, `f` le nom de la fonction et `Tn` le type du paramètre en position `n`.

Appel d'une fonction : `Graph a = dijkstra(g, 1);` ou `g = removeAllPonderation(g);`.

Le passage des paramètres se fait par copie (ou référence constante) et la correspondance est par position. On autorise la surcharge des fonctions.

5.2 Architecture

Un des buts de l'application est de pouvoir travailler sur plusieurs onglets en même temps. Cela implique que chacun aura des variables différentes et indépendantes.

L'approche de base est que chaque onglet aura une instance de l'interpréteur :

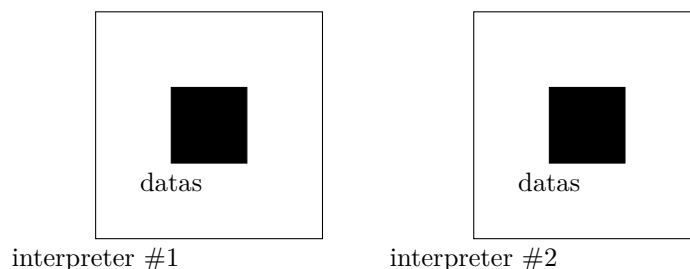


FIGURE 7 – Proposition d'architecture pour l'interpréteur

Cette approche a cependant le désavantage de dupliquer l'interpréteur dans la mémoire, alors que celui-ci est le même. Ce qui définit l'état dans lequel l'interpréteur est, c'est les données dans la mémoire virtuelle de celui-ci, c'est-à-dire la table des variables (voir éventuellement la table des fonctions). On peut donc développer une autre solution, très proche des processeurs et de leurs registres :

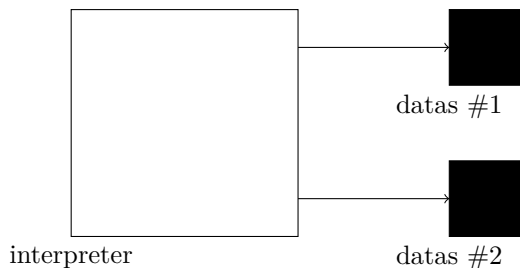


FIGURE 8 – Architecture générale de l'interpréteur

À présent l'interpréteur n'est plus dupliqué, ainsi chaque onglet aura ses données et il suffira d'indiquer à l'interpréteur quel jeu de données il doit utiliser.

5.2.1 Interface utilisateur

Avant de s'attaquer au diagramme de classes, on peut se poser la question de l'interface qui sera fournie à l'utilisateur de l'interpréteur. Très simplement, celui-ci a besoin de :

- Instancier l'interpréteur
- Instancier une ou des tables des variables
- Connecter l'interpréteur à une table des variables
- Interfacer des fonctions avec l'interpréteur (éventuellement)
- Envoyer une commande (requête) à interpréter
- Récupérer le résultat d'une commande
- Connaître les fonctions disponibles
- Connaître les variables disponibles

Notons que les résultats peuvent être de plusieurs types : erreur avec message des détails, succès avec le nom de la variable de résultat.

5.2.2 Diagramme de classes

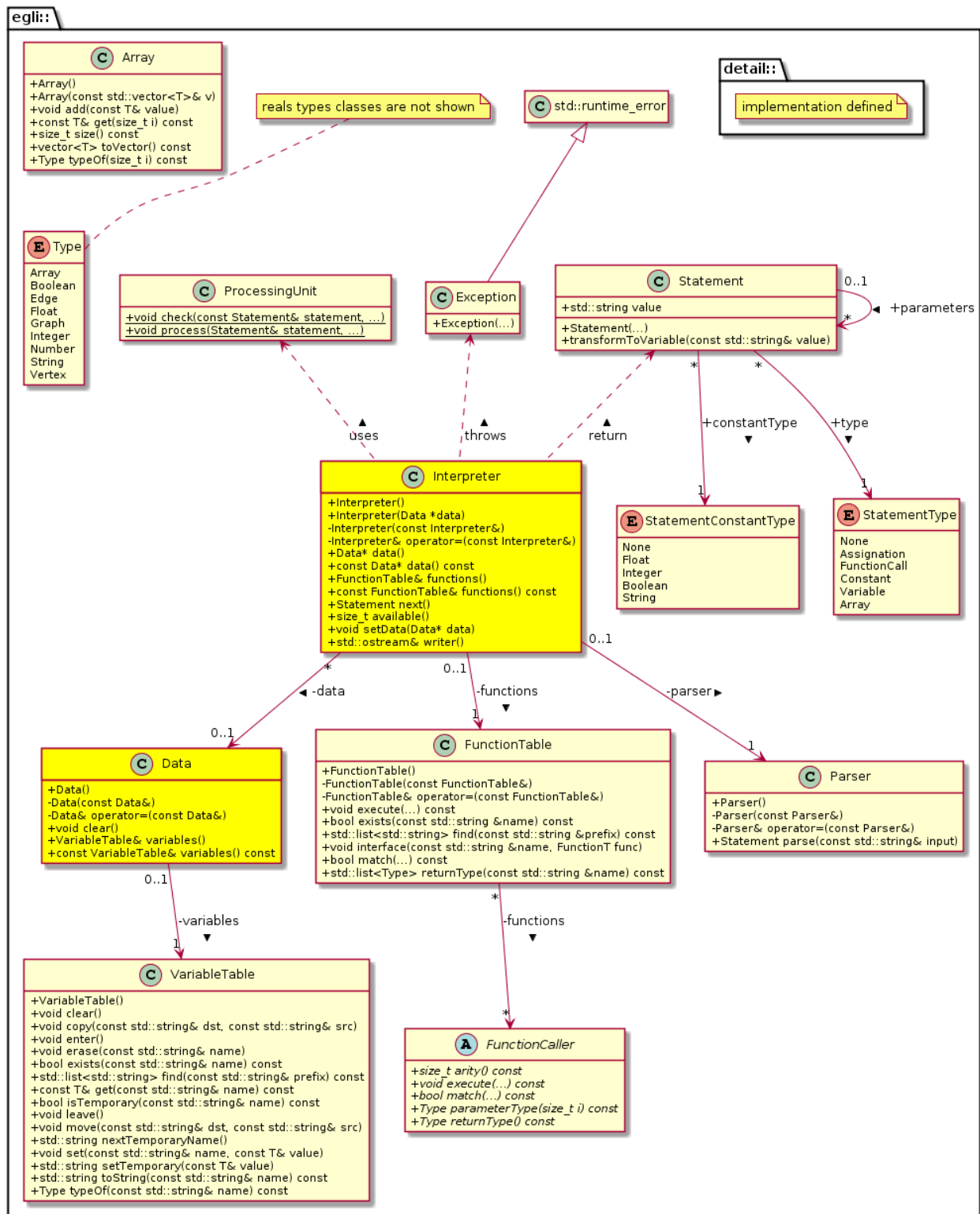


FIGURE 9 – Diagramme de classes de l'interpréteur [3]

Ce diagramme ne montre pas les attributs des classes car l'implémentation de chacune n'est pas fixée. De plus l'interface est là à titre indicative, il est possible qu'il y ait quelques modifications dans les sources. Pour finir certaines classes ont des ..., cela indique que l'on ne sait pas encore à l'heure actuelle ce qu'il y aura à l'intérieur, ou que cela surchargerait trop le diagramme. Notons les classes surlignées en jaune vif, il s'agit des classes principales pour l'utilisateur (voir section 5.3).

5.2.3 Répertoires

L'interpréteur étant une couche indépendante de l'application GUI, nous avons décidé de mettre tout ce qui le concerne dans un `namespace` du nom de `egli`, pour *Embeeded GraphY Language Interpreter*.

Voici à quoi ressemblera le répertoire de l'interpréteur :

```
egli/.....répertoire de base de EGLI
├── detail/.....répertoire "privé", ne doit pas être utilisé par l'utilisateur
│   ├── interface/.....répertoire pour les fonctions interfacées
│   │   ├── algorithms.h.....les algorithmes venant des couches graphes et algorithmes
│   │   ├── basics.h.....les fonctions nécessaires au fonctionnement interne
│   │   ├── builtins.h.....les fonctions offertes de base dans l'interpréteur
│   │   ├── FunctionImpl.h.....implémentation de FunctionCaller
│   │   ├── Grammar.h.....grammaire utilisée par le parseur
│   │   ├── interface.h.....interfaçage centralisé des fonctions
│   │   └── <...>.....autres fichiers d'implémentation
├── Array.h.....tableau dynamique hétérogène
├── Data.h.....données utilisées dans l'interpréteur
├── egli.h.....fichier d'inclusion générale
├── Exception.h.....exception spécifique à EGLI
├── Function.h.....déclaration de FunctionCaller
├── FunctionTable.h.....table des fonctions
├── GraphWrapper.h.....wrapper de IGraph venant des couches graphes et algorithmes
├── Interpreter.h.....interpréteur de EGLI
├── Parser.h.....parseur utilisé par l'interpréteur
├── ProcessingUnit.h.....unité de vérification et de traitement
├── serialize.h.....(dé-)sérialisation de Data
├── Statement.h.....type de traitement possible
├── toString.h.....transforme une variable en std::string
├── VariableTable.h.....table des variables
└── <...>.....autres fichiers
```

5.3 Exemple

Dans cette section, nous allons voir un exemple d'utilisation de EGLI. Un code valant plus qu'un long discours, voici une utilisation en console qui prend les commandes en entrée et qui affiche les sorties correspondantes. On y voit également un exemple d'interfaçage d'une fonction.

```
// main.cpp
// Exemple d'utilisation de EGLI

#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>

// Fichier d'inculsion de EGLI
#include "egli/egli.h"

using namespace std;

// Fonction que l'on souhaite interfacier
int random(int min, int max)
{
    return min + rand() / (RAND_MAX / (max - min + 1) + 1);
}

int main()
{
    srand(time(nullptr));

    // On cree l'interpreteur
    egli::Interpreter interpreter;
```

```
// On interface notre fonction
interpreter.functions().interface("rand", random);

// On a besoin de donnees sur lesquelles travailler
egli::Data data;

// On indique a l'interpreteur quelles donnees il doit utiliser
interpreter.setData(&data);

// On lit l'entree utilisateur
string input;
while (getline(cin, input)) {

    // On quitte si input == "q"
    if (input == "q")
        break;

    // On ecrit l'entree dans l'interpreteur
    // (writer() retourne un std::ostream&)
    interpreter.writer() << input;

    // Tant que des commandes sont disponibles
    while (interpreter.available()) {
        try {

            // On execute la commande en recuperant son resultat
            // (statement.value contiendra le nom de la variable generee)
            egli::Statement statement = interpreter.next();

            // On verifie si la variable existe
            // (elle n'existe pas dans le cas ou c'etait une variable temporaire)
            if (data.variables().exists(statement.value)) {

                // Elle existe! On affiche sa valeur
                string value = data.variables().toString(statement.value);
                string type = egli::toString(data.variables().typeOf(statement.value));
                if (value.size() > 50) { // mise en page
                    value.resize(50);
                    value.append("...");
                }
                cout << ">>" << statement.value << "_" << type << ")_<-_" << value
                     << endl;
            }
            else
                cout << ">>_temporary_returned" << endl;

            // Les erreurs sont gerees grace aux exceptions
        } catch(const exception &ex) {
            cout << ex.what() << endl;
        }
    }

    cout << endl; // mise en page
}

// Fin
return 0;
}
```

5.3.1 Fonctions incluses

Cette section recense les fonctions incluses dans l'interpreteur et disponibles pour l'utilisateur de l'application. Les fonctions *built-in* viennent de l'interpreteur directement, les fonctions *algo* proviennent de la couche sur les graphes et ses algorithmes, et les fonctions GUI sont spécifiques à la couche interface utilisateur.

Note : les paramètres de type `T` indiquent qu'il existe une surcharge de la fonction pour chacun des types disponible dans le langage.

Prototype	Description	Origine
<code>String toString(T a)</code>	Retourne la représentation de <code>a</code> sous forme de chaîne	<i>built-in</i>
<code>Boolean save(Graph g, String file)</code>	Sauvegarde le graphe <code>g</code> dans le fichier <code>file</code>	<i>built-in</i>
<code>Graph load(String file)</code>	Charge le graphe contenu dans le fichier <code>file</code>	<i>built-in</i>
<code>String typeOf(T a)</code>	Retourne le type de <code>a</code> sous forme de chaîne	<i>built-in</i>
<code>Graph er(Integer V, Float p)</code>	Génère un graphe aléatoire à <code>V</code> sommets et une probabilité d'inclusion des arêtes de <code>p</code>	<i>built-in</i>
<code>Boolean draw(Graph g)</code>	Dessine le graphe <code>g</code>	<i>GUI</i>
<code>Boolean exportAsSvg(Graph g)</code>	Exporte le graphe <code>g</code> en SVG	<i>GUI</i>
<code>Boolean exportAsSvg(Graph g, String file)</code>	Exporte le graphe <code>g</code> en SVG dans le fichier <code>file</code>	<i>GUI</i>
<code>Array bellmanFord(Graph g, Integer from)</code>	Applique un Bellman-Ford au graphe <code>g</code> depuis le sommet <code>from</code>	<i>algo</i>
<code>Array bfs(Graph g, Integer from)</code>	Applique un BFS sur le graphe <code>g</code> depuis le sommet <code>from</code>	<i>algo</i>
<code>Array cc(Graph g)</code>	Cherche les composantes connexes du graphe <code>g</code>	<i>algo</i>
<code>Graph detectCycle(Graph g)</code>	Retourne un cycle du graphe <code>g</code> , si il y en a un	<i>algo</i>
<code>Array dfs(Graph g, Integer from)</code>	Applique un DFS sur le graphe <code>g</code> depuis le sommet <code>from</code>	<i>algo</i>
<code>Array dijkstra(Graph g, Integer from)</code>	Applique un Dijkstra au graphe <code>g</code> depuis le sommet <code>from</code>	<i>algo</i>
<code>Boolean isConnected(Graph g)</code>	Vérifie si le graphe <code>g</code> est connexe	<i>algo</i>
<code>Boolean isDirected(Graph g)</code>	Vérifie si le graphe <code>g</code> est orienté	<i>algo</i>
<code>Boolean isEmpty(Graph g)</code>	Vérifie si le graphe <code>g</code> est vide	<i>algo</i>
<code>Boolean isNegativeWeighted(Graph g)</code>	Vérifie si le graphe <code>g</code> possède des poids négatifs	<i>algo</i>
<code>Boolean isNull(Graph g)</code>	Vérifie si le graphe <code>g</code> est nul	<i>algo</i>
<code>Boolean isPlanar(Graph g)</code>	Vérifie si le graphe <code>g</code> est planaire	<i>algo</i>
<code>Boolean isSimple(Graph g)</code>	Vérifie si le graphe <code>g</code> est simple	<i>algo</i>
<code>Boolean isStronglyConnected(Graph g)</code>	Vérifie si le graphe <code>g</code> est fortement connexe	<i>algo</i>
<code>Boolean isWeighted(Graph g)</code>	Vérifie si le graphe <code>g</code> est pondéré	<i>algo</i>
<code>Graph kruskal(Graph g)</code>	Applique un Kruskal au graphe <code>g</code>	<i>algo</i>
<code>Array tarjan(Graph g)</code>	Applique un Tarjan au graphe <code>g</code>	<i>algo</i>
<code>Array topologicalSort(Graph g)</code>	Applique un tri topologique au graphe <code>g</code>	<i>algo</i>

6 Graphes

6.1 Préface

Un graphe est une structure composée d'un ensemble de sommets et d'arêtes/arcs. Il existe différents moyens de les représenter.

Cette section a pour but de présenter les interfaces à disposition de l'utilisateur pour manipuler des graphes, ainsi que de montrer nos choix de modélisation et d'implémentation de ceux-ci.

6.2 Architecture

6.2.1 Diagramme de classe

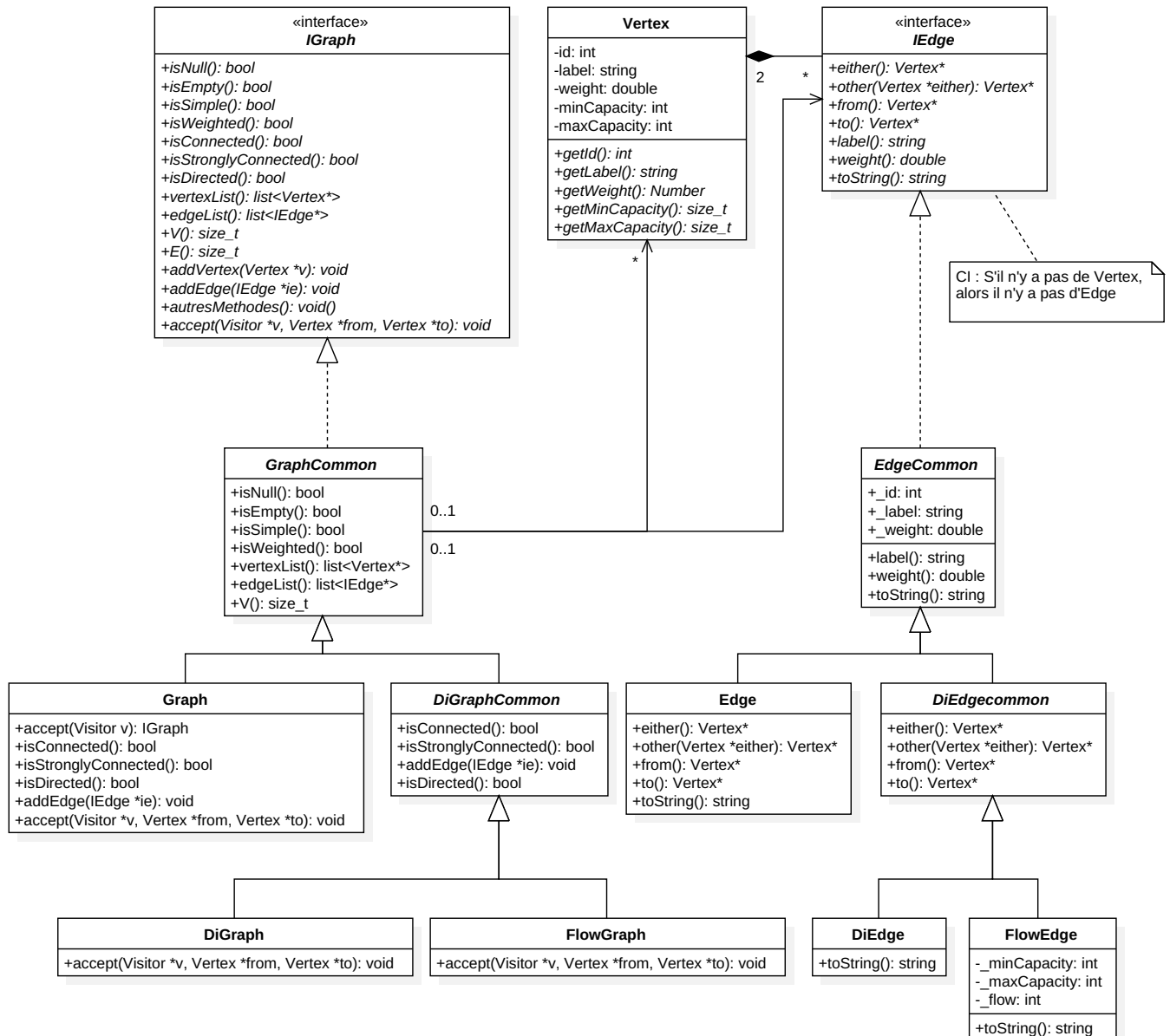


FIGURE 10 – Diagramme de classe pour les graphes

6.2.2 La classe "Vertex"

Les sommets sont les mêmes pour tous les types de graphe (orienté, à flot, ou non-orienté). Les informations suivantes y sont stockées :

- un identifiant géré par la classe "Graph" ci-après
- un label pour spécifier un nom au sommet
- un poids de type numérique qui pourra être ensuite redistribué sur les arêtes par exemple
- une capacité minimale de type numérique entière
- une capacité maximale de type numérique entière

Voici un aperçu des attributs de la classe en question

```
// Aperçu de la classe Vertex
public class Vertex {
    private:
        int _id;
```

```

    string _label;
    double _weight;
    int _minCapacity;
    int _maxCapacity;

public:
    // Constructeur vide. Valeurs par défaut.
    Vertex() : _id(-1), _label(""), _weight(numeric_limits<double>::max()),
               _minCapacity(-1), _maxCapacity(-1) {}
    // setter et getter...
    // ...
};

```

Les capacités sont des entiers signés uniquement pour leur donner une valeur par défaut de -1. On aurait pu les déclarer avec un type non-signé mais nous trouvions plus léger au niveau du code de le faire de cette manière.

6.2.3 La classe "Edge"

Les arcs/arêtes sont plus délicats car ils diffèrent entre les types de graphe. Les trois types présents dans l'application sont les suivants :

- arête, type `Edge`
- arc, type `DiEdge` pour Directed Edge
- arc à flot, type `FlowEdge`

Les attributs communs aux trois types sont :

- un id pour identifier l'Edge de façon unique au sein d'un graphe donné
- un pointeur sur le sommet source nommé *a*
- un pointeur sur le sommet destination nommé *b*
- un label pour nommer l'Edge
- un poids de type numérique pour lui associer un poids
- En plus pour les `FlowEdge` : une capacité minimum de type numérique entière
- En plus pour les `FlowEdge` : une capacité maximum de type numérique entière
- En plus pour les `FlowEdge` : le flot courant de type numérique entier

Notons que pour un `Edge`, les sommets *a* et *b* ne font pas office de "source" et "destination" puisque l'arc n'est pas orienté.

Nous sommes donc partis sur une première approche qui consiste à avoir une classe `Edge` commune dont les `Edge` spécifiques hériteront selon le type de graphe.

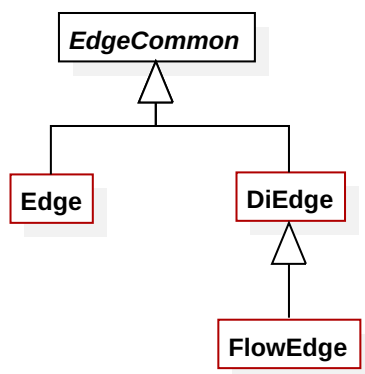


FIGURE 11 – Classe Edge solution 1

Cette approche est optimale car elle permet une bonne factorisation du code. Seulement, nous avons rencontré un problème lors de la mise en place de la classe `Graph` où les types de `Edge` sont passés par template (expliqué dans le chapitre "classe `Graph`"). Avec la solution ci-dessus nous n'avons pas trouvé le moyen de spécifier explicitement le type `FlowEdge` au graphe sans générer d'erreur de compilation : `FlowEdge` étant un sous-type

de `DiEdge`, le compilateur ne parvenait pas à faire le lien sur le bon type. Nous nous sommes donc résiliés à modifier l'architecture de la classe "Edge" pour n'avoir que des types distincts.

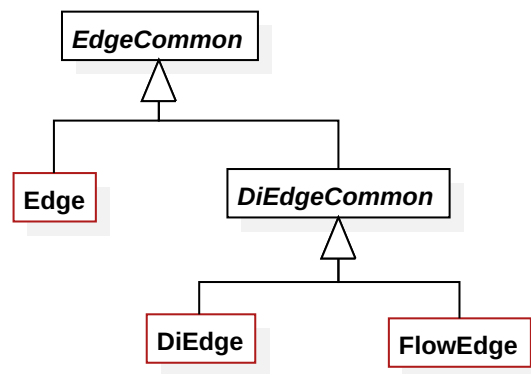


FIGURE 12 – Classe Edge solution 2

Une interface `IEdge` est ensuite ajoutée pour fournir à l'utilisateur un moyen unique de manipuler tout type de Edge.

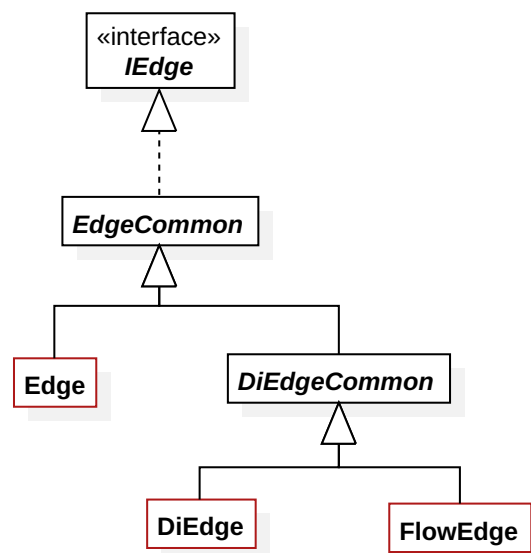


FIGURE 13 – Classe Edge solution finale

Ainsi, grâce à la covariance des variables, l'utilisateur peut voir n'importe quel type de Edge en tant qu'`IEdge` et le manipuler de la sorte.

```

// Edge entre le sommet v1 et v2
IEdge *edge = new Edge(v1, v2, 4.5);

// DiEdge du sommet v1 au sommet v2
IEdge *diEdge = new DiEdge(v1, v2, 6.0);

// FlowEdge de capacite 3 du sommet v1 au sommet v2
IEdge *flowEdge = new FlowEdge(v1, v2, 5.2, 3);

// exemple de methode
cout << edge->weight() << endl; // Affiche 4.5
cout << diEdge->weight() << endl; // Affiche 6
cout << flowEdge->weight() << endl; // Affiche 5.2
  
```

6.2.4 Classe "Graph"

Cette classe est étroitement liée à la classe "Edge". Sachant qu'il existe trois types d'Edge, il existe donc 3 types de graphe :

- Graphe non-orienté
- Graphe orienté
- Graphe orienté à flot

Nous gardons la même architecture que la classe "Edge". Les trois types doivent être distincts (un graphe à flot ne doit pas hériter directement d'un graphe orienté) pour expliciter clairement le type de Edge. Une première approche nous avait mené à déclarer les classes "Edge" internes aux classes "Graph" correspondantes mais cette méthode s'est avérée trop complexe. Une deuxième approche consistait plutôt à passer le type de Edge en template à la classe Graph correspondante. Cette deuxième approche nous paraissait plus simple et nous avons donc opté pour celle-ci. Voici à quoi ressemble le modèle conceptuel de la classe "Graph".

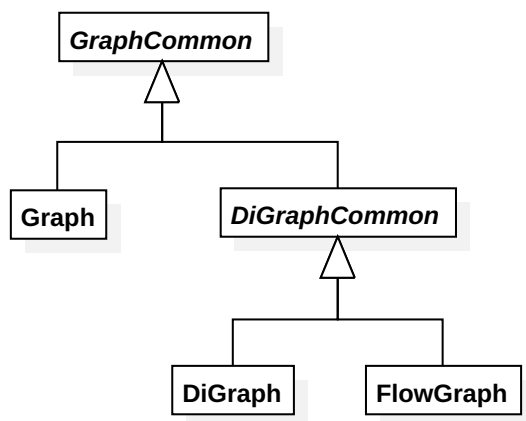


FIGURE 14 – Classe Graph solution finale

Et voici comment expliciter le type de Edge avec les templates.

```
template <typename T>
class GraphCommon {
    //...
}

template <typename T>
class DiGraphCommon : public GraphCommon<T> {
    //...
}

class Graph : public GraphCommon<Edge> {
    //...
}

class DiGraph : public DiGraphCommon<DiEdge> {
    //...
}

class FlowGraph : public GraphCommon<FlowEdge> {
    //...
}
```

Ainsi, **Graph** contient des **Edge**, **DiGraph** contient des **DiEdge**, et **FlowGraph** contient des **FlowEdge**. Lors d'un appel à une méthode implémentée dans **GraphCommon** qui traite des **IEdge**, la méthode effectuera le lien sur le bon type de Edge au travers de la liaison dynamique.

Pour abstraire tout ce contenu à l'utilisateur, une interface lui permettant la manipulation de tout type de graphes est créée au-dessus de l'architecture actuelle. Voici une ébauche de cette interface permettant la manipulation de graphes :

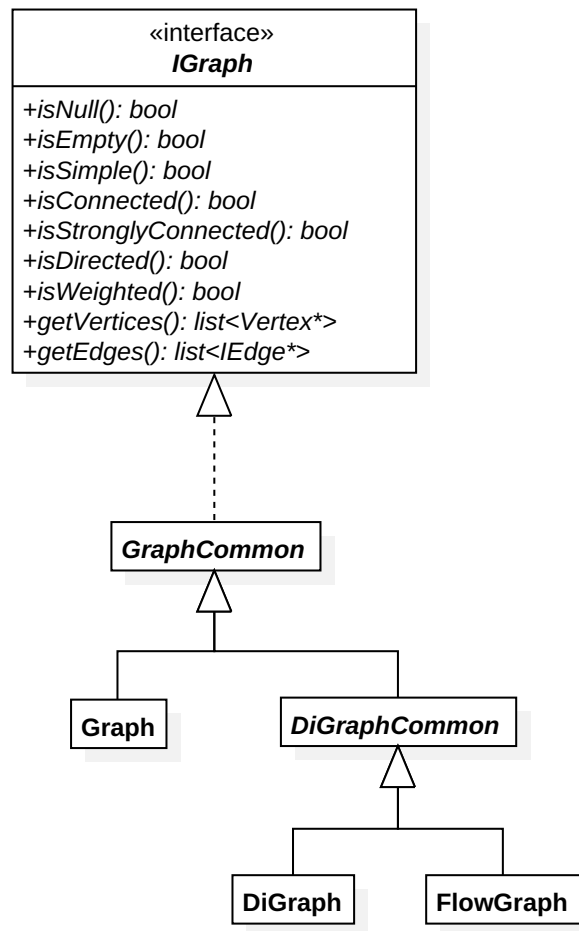


FIGURE 15 – Interface IGraph

Cette interface fournit les méthodes nécessaires aux utilisateurs pour manipuler n'importe quel type de graphe. La création d'un graphe se fait alors de la manière suivante :

```
// Creation d'un graphe non-orienté vide (sans sommet)
IGraph *graph = new Graph;

// Creation d'un graphe orienté vide (sans sommet)
IGraph *diGraph = new DiGraph;

// Creation d'un graphe à flot vide (sans sommet)
IGraph *flowGraph = new FlowGraph;

// exemple de méthode
cout << graph->isDirected() << endl; // Affiche false
cout << diGraph->isDirected() << endl; // Affiche true
cout << flowGraph->isDirected() << endl; // Affiche true
```

6.2.5 Structure de donnée de la classe "Graph"

On peut distinguer deux approches principales pour représenter et stocker un graphe :

1. à l'aide de matrices
2. ou à l'aide de listes (ou tableaux)

L'utilisation de matrices est prépondérante dans la modélisation mathématique de nombreux problèmes et l'étude de certaines propriétés d'un graphe. Elle permet des vérifications rapides pour la présence ou non d'arcs/arêtes, mais est relativement lente lorsqu'il s'agit de les parcourir.

Quant aux listes, notamment les listes d'adjacence, l'itération des arcs/arêtes est rapide mais la vérification de leur présence est lente. Les listes sont un moyen compact de représenter les matrices car elles stockent uniquement les arêtes existantes. Le gain de place mémoire est donc plus avantageux tant que le graphe n'est pas

complet.

À ce stade, le choix de l'un ou l'autre présente alors ses avantages et désavantages. Une petite étude nous a permis d'orienter notre choix :

Une première approche simple nous permet de déterminer qu'un graphe stocké sous forme de matrice occupera n^2 octets, où n est le nombre de sommet. Dans ce cas, un type char ferait amplement l'affaire pour représenter chaque case, car en non-signé, on pourrait avoir jusqu'à 255 arcs/arêtes entre deux sommets.

$$\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & v_5 \\ 0 & 2 & 4 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 & 1 \\ 1 & 1 & 0 & 3 & 0 \end{bmatrix}$$

FIGURE 16 – Matrice d'adjacence de char

La seconde approche vise à réduire la taille mémoire de façon à n'avoir qu'un seul bit par case pour la présence ou non d'une arête. Cette approche économise huit fois plus de mémoire mais est inutilisable lorsqu'il y a plus de deux arêtes entre deux sommets.

$$\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & v_5 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

FIGURE 17 – Matrice d'adjacence de bit

Ces deux premières approches ne permettent pas de stocker beaucoup d'informations concernant les Edge, c'est pourquoi nous nous tournons vers une troisième approche qui consiste à stocker des pointeurs de [IEdge](#), ou nullptr lorsque l'arc/arête n'est pas présent. Pour pouvoir stocker plusieurs Edge, chaque case contient un pointeur vers une collection d'objets [IEdge](#). Cette troisième approche nécessite néanmoins 32 bits par case donc une taille mémoire de $4 * n^2$ octets.

$$\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & v_5 \\ nullptr & *edges & nullptr & *edges & *edges \\ nullptr & nullptr & nullptr & nullptr & nullptr \\ *edges & nullptr & nullptr & nullptr & *edges \\ nullptr & nullptr & nullptr & nullptr & *nullptr \\ *edges & *edges & nullptr & nullptr & nullptr \end{bmatrix}$$

FIGURE 18 – Matrice d'adjacence de collection d'Edge

Cependant, après avoir suivi le cours de GRE, nous nous sommes rendus compte qu'il était rare d'obtenir des graphes complets. La majorité des cas d'utilisation requièrent des graphes de faible densité, ce qui nous amène vers une quatrième approche : stocker les graphes sous forme de liste d'adjacence.

Mis à part les aspects liés au gain de mémoire, les listes d'adjacence facilitent la plupart des opérations présentes dans nos algorithmes. Trouver tous les sommets adjacents à un certain sommet est aussi simple que

lire la liste et prend un temps proportionnel au nombre de voisins. Avec une matrice, toute une ligne doit être lue, ce qui prend un temps beaucoup plus long, égal au nombre de sommets présents dans le graphe entier.

Cette quatrième et dernière approche est celle que nous avons choisie car elle est la plus adaptée à nos besoins. Nos graphes sont donc stockés sous forme de liste d'adjacence.

6.2.6 Détails d'implémentation de la structure

Dans cette partie, nous définissons comment la liste d'adjacence est implémentée. Il existe trois manières de représenter une telle liste :

1. Pour chaque sommet, lui associer un tableau des sommets adjacents. Il n'y a pas de représentation explicite des arcs/arêtes.
2. Un tableau indexé par les numéros de sommet, où chaque case contient une simple liste chaînée constituée des sommets adjacents, ou des arcs/arêtes adjacent(e)s.
3. Une approche plus orientée objet consiste à avoir une variable d'instance pour chaque sommet, pointant vers une collection d'objets qui liste les arcs/arêtes voisin(e)s. À l'opposé, chaque arc/arête pointe vers les sommets qui la compose.

La solution (1) ne permet pas de stocker d'informations sur les arcs/arêtes. La solution (2) le permet, pour autant qu'on choisisse une liste d'Edge adjacents. Dans ce cas, on ne peut pas stocker d'information sur les sommets.

Nous avons donc opté pour la solution (3). Un tableau de sommets, où chaque sommet possède une variable pointant vers une liste de pointeurs de `IEdge`, et où chaque `IEdge` possède un pointeur sur chacun des sommets qui la constitue. Nous aurions également pu stocker des pointeurs de `T` (`T` étant le typename du template, donc le type de l'Edge) mais pour avoir une visibilité plus globale sur les Edge, nous avons décidé de stocker des pointeurs de `IEdge`.

À noter que pour un graphe non orienté, les Edge se répéteront à double et c'est précisément ce que nous voulons car nous souhaitons préserver la logique de la liste d'adjacence pour nos algorithmes.

Le choix de stocker les éléments par pointeur ou par copie a été longuement réfléchi. L'un comme l'autre a ses avantages et désavantages.

	Avantages	Désavantages
Copie	On peut passer le même Vertex déclaré une seule fois à plusieurs graphes. Même chose pour les Edge.	Prend beaucoup de place mémoire, surtout que la structure peut vite devenir grande. Un vertex ne peut pas être modifié en dehors de la structure (peut aussi être un avantage...)
Pointeur	Le sommet ou l'edge manipulé n'existe qu'une seule fois et peut être modifié depuis n'importe où (peut être dangereux)	Nécessite de gérer soi-même la désallocation de mémoire

L'argument de l'espace mémoire alloué a porté notre choix final sur l'utilisation de pointeurs pour stocker les éléments.

Voici un exemple de liste d'adjacence. `*e{chiffre}` est un pointeur sur l'Edge `{chiffre}`. Les chiffres en marge à gauche sont les index des sommets.

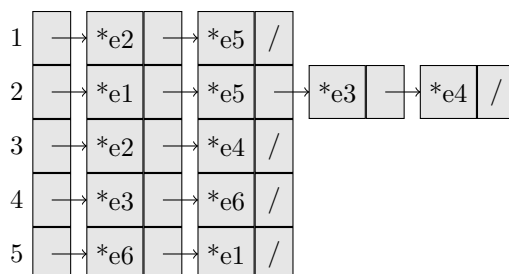


FIGURE 19 – Liste d'adjacence

Dans cet exemple :

- le sommet 1 possède deux Edge adjacents e2 et e5.
- le sommet 2 possède quatre Edge adjacents e1, e5, e3, e4.

- le sommet 3 possède deux Edge adjacents e2 et e4.
- etc.

Et voici à quoi ressemble le code de cette structure. À noter que nous avons besoin de garder un tableau des sommets.

```
template <typename T>
class GraphCommon : public IGraph {
private:
    vector<Vertex*> _vertices;
    vector<list<IEdge*>> _adjacentList;
public:
    template <typename Func>
    void forEachAdjacentEdge(Vertex v, Func f) {
        for (Edge* e : adjacentList[v->getId()])
            f(e); // operation sur les edges adjacents au sommet v
    }
};
```

6.2.7 Création de graphes

La création de graphe peut maintenant s'effectuer de la manière suivante :

```
// Creer un graphe non oriente
public static void main() {
    // creer quelques sommets et edges
    Vertex *v1 = new Vertex;
    Vertex *v2 = new Vertex;
    Vertex *v3 = new Vertex;
    IEdge *e1 = new Edge(v1, v2);
    IEdge *e2 = new Edge(v1, v3);

    vector<Vertex*> vertices = {v1, v2, v3};
    vector<IEdge*> edges = {e1, e2};

    // Creation d'un graphe non oriente
    IGraph *myGraph = new Graph(vertices, edges);

    // Utilisation d'une methode de l'interface
    if (myGraph->isSimple())
        cout << "myGraph est simple" << endl;
}
```

```
// Creer un graphe oriente
public static void main() {
    // creer quelques sommets et edges
    Vertex *v1 = new Vertex;
    Vertex *v2 = new Vertex;
    Vertex *v3 = new Vertex;
    IEdge *e1 = new DiEdge(v1, v2);
    IEdge *e2 = new DiEdge(v1, v3);

    vector<Vertex*> vertices = {v1, v2, v3};
    vector<IEdge*> edges = {e1, e2};

    // Creation d'un graphe oriente
    IGraph *myGraph = new DiGraph(vertices, edges);

    // Utilisation d'une methode de l'interface
    if (myGraph->isSimple())
        cout << "myGraph est simple" << endl;
}
```

```
// Creer un graphe oriente a flot
public static void main() {
```

```

// creer quelques sommets et edges
Vertex *v1 = new Vertex;
Vertex *v2 = new Vertex;
Vertex *v3 = new Vertex;
IEdge *e1 = new FlowEdge(v1, v2);
IEdge *e2 = new FlowEdge(v1, v3);

vector<Vertex*> vertices = {v1, v2, v3};
vector<IEdge*> edges = {e1, e2};

// Creation d'un graphe oriente a flot
IGraph *myGraph = new FlowGraph(vertices, edges);

// Utilisation d'une methode de l'interface
if (myGraph->isSimple())
    cout << "myGraph est simple" << endl;
}

```

6.3 Algorithmes

6.3.1 Pattern Visitor

Le patron de conception Visiteur permet d'appliquer efficacement un algorithme à un graphe. L'utilisation est la suivante :

- L'algorithme est le visiteur
- Le graphe représente la structure de donnée à visiter

L'interface Visitor peut visiter un [Graph](#), un [DiGraph](#), ou un [FlowGraph](#) et ce en spécifiant un sommet source "from" et un sommet destination "to". Tous nos algorithmes n'utiliseront pas "from" et "to", c'est pourquoi ils ont une valeur par défaut à nullptr.

Le résultat de nos algorithmes retournent parfois un graphe, parfois un tableau. Deux méthodes supplémentaires `IGraph *G();` et `vector<double> table();` ont donc été ajoutées.

Du côté de la structure, une méthode `accept(Visitor *v, Vertex *from, Vertex *to)` permet de spécifier au visiteur quel type de graphe manipuler au travers de la liaison dynamique.

Finalement, une classe statique `GraphAlgorithm` liste tous les algorithmes implémentés (voir dans la section "liste des algorithmes implémentés").

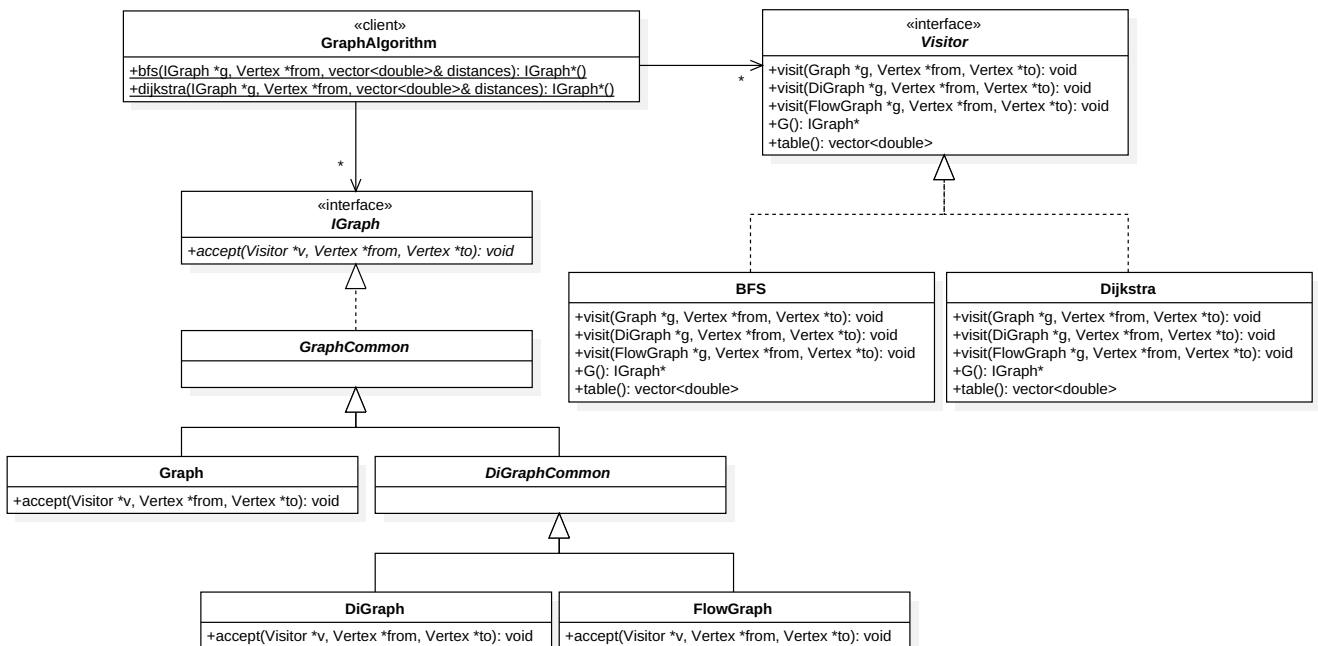


FIGURE 20 – Pattern Visitor pour l'application des algorithmes aux graphes

Les algorithmes ne sont pas les mêmes pour tous les types de graphe : un parcours en largeur sera quelque peu différent entre un graphe orienté et non-orienté. Le pattern Visiteur est précisément fait pour répondre à

ce genre de problème. De plus, l'avantage du pattern est qu'il est facilement extensible : un nouvel algorithme s'ajoute tout bonnement en tant que visiteur. Par contre, la structure de donnée doit être complète et tout changement de cette dernière demandera, selon le changement, une adaptation chez tous les visiteurs.

```
// Code d'un objet visite
class Graph {
private:
    //...
public:
    void accept(Visitor *v, Vertex *from, Vertex *to) {
        v->visitGraph(this, from, to);
    }
};
```

```
// Implementation d'un visiteur
class BFS : public Visitor {
private:
    vector<double> _distances;
    IGraph *_G;

public:
    BFS() : _distances(0), _G(nullptr) {}
    void visit(Graph *g, Vertex *from, Vertex *to) {
        // Implementation du BFS pour un graphe non-orienté...
    }
    void visit(DiGraph *g, Vertex *from, Vertex *to) {
        // Implementation du BFS pour un graphe orienté...
    }
    void visit(FlowGraph *g, Vertex *from, Vertex *to) {
        // Implementation du BFS pour un graphe a flot...
    }
    IGraph* G() const {
        return _G;
    }
    vector<double> table() {
        return _distances;
    }
};
```

```
// Implementation de deux algorithmes dans GraphAlgorithm
#include "BFS.h"
#include "Dijkstra.h"

class GraphAlgorithm {
public:
    IGraph *bfs(IGraph *g, Vertex *from, vector<double>& distances) {
        Visitor *v = new BFS;
        g->accept(v, from);
        distances = v->table();
        return v->G();
    }

    IGraph *dijkstra(IGraph *g, Vertex *from, vector<double>& distances) {
        Visitor *v = new DijkstraSP;
        g->accept(v, from);
        distances = v->table();
        return v->G();
    }
};
```

Nous avons choisi ce pattern car il permet de bien séparer la structure de donnée des algorithmes et il nous permet de collaborer efficacement au sein de l'équipe.

6.3.2 Liste des algorithmes implémentés

Dans le tableau ci-dessous figurent les algorithmes traités par l'application. Pour la colonne *Complexité*, V représente le nombre de sommet (Vertex), et E le nombre d'arcs/arêtes (Edge) d'un graphe.

Nom	Valeur de retour	Description	Complexité
BFS	IGraph	Breadth First Search : effectue une exploration en largeur d'un graphe depuis un sommet source et retourne l'arborescence du parcours.	$O(V + E)$
DFS	IGraph	Depth First Search : effectue une exploration en profondeur d'un graphe depuis un sommet source et retourne l'arborescence du parcours.	$O(V + E)$
Composante connexe	vector<double>	Calcule les composantes connexes d'un graphe et retourne un tableau dont l'index représente le sommet et la valeur la composante à laquelle appartient le sommet.	$O(V + E)$
Composante fortement connexe	vector<double>	Calcule les composantes fortement connexes d'un graphe orienté et retourne un tableau dont l'index représente le sommet et la valeur de la composante à laquelle appartient le sommet.	$O(V + E)$
Kruskal	IGraph et vector<double> en paramètre	Implémenté avec UnionFind, recherche de l'arbre recouvrant de poids minimum.	$O(E * \log(V))$
Bellman-Ford	IGraph et vector<double> en paramètre	Recherche les plus courts chemins depuis un sommet source vers tous les autres sommets. Le graphe doit être orienté et pondéré.	$O(E * V)$
Dijkstra	IGraph et vector<double> en paramètre	Implémenté à l'aide d'une queue de priorité, recherche d'un plus court chemin depuis un sommet source vers tous les autres sommets. Le graphe doit être orienté et pondéré positivement (ou nul).	$O(E * \log(V))$
Détection des cycles	IGraph	Détermine si un graphe contient un cycle ou un circuit pour les graphes orientés et retourne le cycle trouvé.	$O(E + V)$
Tri Topologique	vector<double>	recherche l'ordre par lequel il faut parcourir le graphe topologiquement. Le graphe doit être orienté ou à flow.	$O(E + V)$

6.3.3 Utilisation

L'utilisation se fait de la manière suivante

1. Créer un graphe (avec des sommets et des edges)
2. Appeler l'algorithme voulu
3. Récupérer le graphe résultant de l'algorithme (si l'algorithme retourne un graphe), sinon récupérer le tableau de double (et éventuellement le tableau de doubles passé préalablement en paramètre)

```
// Exemple d'utilisation
#include "Graph.h"
#include "GraphAlgorithm.h"

int main {
    // Creation d'un graphe non orienté
    Vertex *v1 = new Vertex;
    Vertex *v2 = new vertex;
    Vertex *v3 = new vertex;

    IEdge *e1 = new Edge(v1, v2);

    IGraph *g = new Graph;

    g->addVertex(v1);
    g->addVertex(v2);
    g->addVertex(v3);
```

```
g->addEdge(e1);

// Application d'un BFS depuis v1
vector<double> distances;
IGraph *bfs = GraphAlgorithm::bfs(g, v1, distances);

// Resultat
cout << *bfs << endl;    // Affiche v1--v2
for (double d : distances)
    cout << d << " ";    // Affiche 0 1 -1
cout << endl;

return EXIT_SUCCESS;
};
```

7 Conclusion

7.1 Application fournie

Vis à vis du cahier des charges initial (disponible en annexe), l'application répond à toutes les fonctionnalités prévues hormis l'implémentation de certains algorithmes, dont voici la liste :

- Arborescence recouvrante de poids min
- Arborescence recouvrante de section min/max
- Flot de capacité fixée (ou min)
- Flot de capacité fixée (ou min) de poids min

Les fonctionnalités optionnelles et futures n'ont pas pu être intégrées par manque de temps, cependant leur ajout est tout à fait réalisable car la conception générale de l'application a été prévue pour être ouverte aux extensions. Notons encore que l'interface graphique n'est pas tout à fait la même que la maquette présente dans le cahier des charges, néanmoins les fonctionnalités sont les mêmes.

Concernant la stabilité du programme, la majorité des bugs/crashes découverts pendant le projet a été corrigée. Cependant certains cas posent toujours problème :

1. Un graphe ne possédant pas tous les sommets dans l'ordre (ex : `g=dfs({#2}, 1);`) pose problème à certaines fonctionnalités (ex : `draw(g);`, `egli::deserialize(...);`)
2. Fuites mémoires détectées au niveau de la couche *graphes et algorithmes*
3. La fenêtre d'aide ne se ferme pas lorsque que l'on quitte la fenêtre principale (avec la croix en haut à droite)
4. La grammaire du langage ne permet pas de tout faire (ex : `v=(0); g={v};, v=(0:-1);, draw(dfs(g, 0)[0]);`)

Les cas (1) et (2) sont des priorités absolues pour la suite du développement, mais nous les avons malheureusement découverts trop tard pour pouvoir les corriger en l'état. Par chance, les causes des problèmes sont connues, ce qui permettra de les corriger rapidement. De plus, nous avons découvert l'existence du logiciel DrMemory [8] qui nous offre la possibilité de cibler les fuites mémoires très précisément.

Un petit mot sur le langage développé spécifiquement pour l'application. Celui-ci demande un petit temps d'apprentissage, mais la prise en main est assez rapide. L'aide intégrée et ses pages bien expliquées (facilement extensible), sa fonction de recherche par mots-clés, ainsi que l'auto-complétion des fonctions et variables prend l'utilisateur par la main et rend l'utilisation de l'application agréable.

Au final, nous sommes satisfaits du résultat final bien qu'il reste encore d'importants bugs à corriger. Le projet offre de nombreuses possibilités d'amélioration, tant au niveau de l'interface graphique que du reste, et nous sommes très contents de cela. Voici une liste non-exhaustive des améliorations possibles auxquelles nous avons pensé :

- Meilleure visualisation des graphes
- Affichage des variables créées dans une zone de l'interface graphique
- Coloration syntaxique
- Fonctionnalités optionnelles et futures du cahier des charges
- ... et plein d'autres

7.2 Planification

La planification initiale du projet (disponible en annexe) a fait ressortir plusieurs problèmes. Premièrement, les temps d'analyse et de conception ont été très largement sous-estimés, ce qui a engendré un gros décalage des tâches :

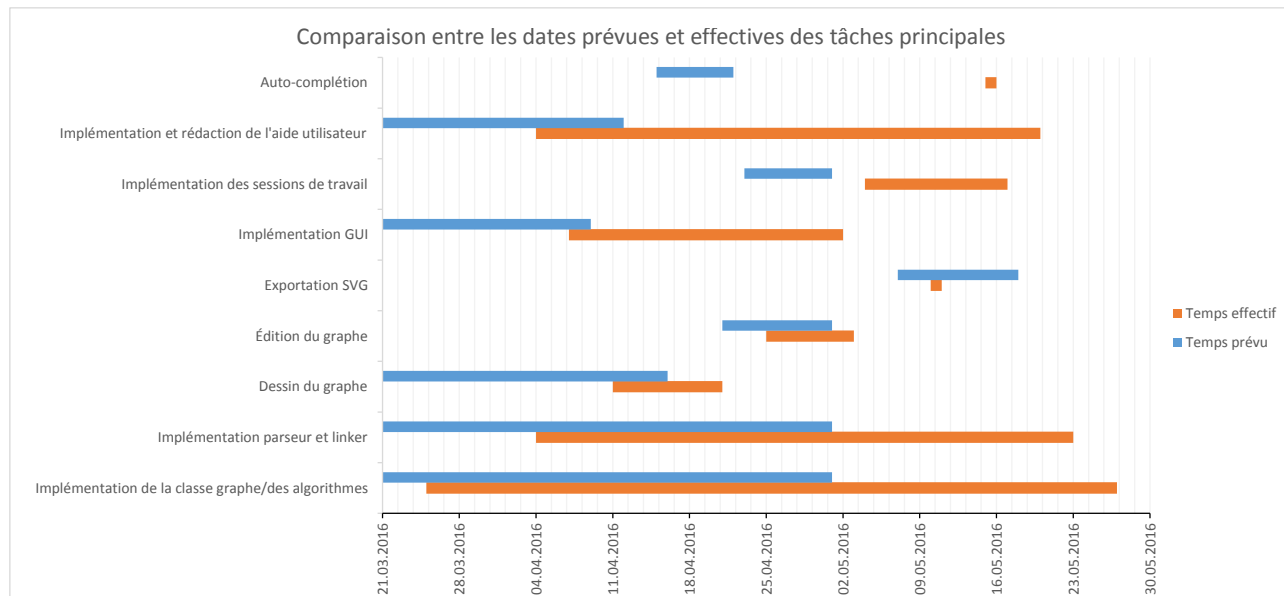


FIGURE 21 – Comparaison entre les dates prévues et effectives des tâches principales

On peut expliquer cela par deux choses : certains problèmes étaient plus complexes que prévus, et l'apprentissage de Qt était beaucoup plus long. En effet, Qt est très puissant et offre de multiples manières d'aborder un problème, en contre-partie cela induit des pertes de temps lorsque l'on part dans la mauvaise direction. Cependant une fois que Qt est pris en main, les gains lors de l'implémentation sont conséquents.

Deuxièmement, la planification initiale était une suite de tâches à faire complètement avant de passer à la suivante. Dans les faits, les tâches (pour une même personne) ont été réalisées en parallèle, cela a impliqué une dissonance entre les temps prévus et les temps effectifs.

Troisièmement, la partie sur les graphes et les algorithmes a été la plus largement sous-estimée, le graphique suivant le montre assez clairement :

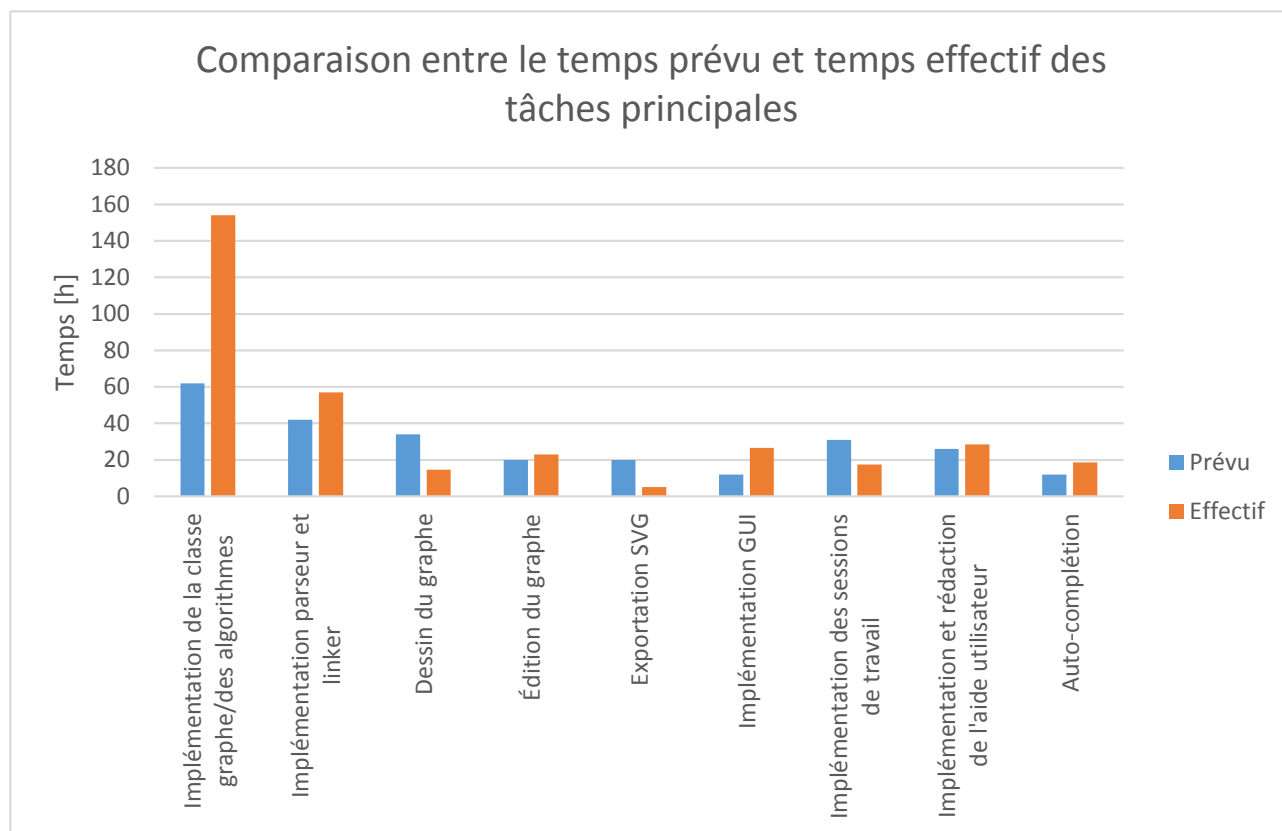


FIGURE 22 – Comparaison entre le temps prévu et temps effectif des tâches principales

Pour finir, les différentes tâches, bien que mal planifiées, ont été bien cernées dès le début du projet. Cela a permis une séparation du travail correcte et indépendante, ainsi qu'une application finale utilisable. Nous sommes cependant convaincus qu'utiliser un modèle de développement itératif (et non pas en cascade) aurait apporté un plus au projet. En effet, cela nous aurait obligé à mettre les différentes couches en commun plus souvent, et ainsi détecter les problèmes avant la fin du projet.

7.3 Travail de groupe

Le principal problème, comme mentionné dans la section précédente, a été le fait que nous n'avons pas fixé suffisamment de délais intermédiaires et que la mise en commun n'a été faite qu'à la fin du projet. Cela a posé des problèmes que nous n'avons malheureusement pas pu résoudre dans les temps.

Le second problème a été un laxisme en début de semestre, nous avons donc dû fournir un gros effort lors des dernières semaines. Lors de ces "rushs", où nous avons travaillé tous ensemble dans la même pièce, il a été intéressant de constater la grande efficacité avec laquelle nous avons travaillé. En effet les problèmes se résolvent beaucoup plus vite et l'ambiance offre un cadre de travail beaucoup plus motivant et encourageant. Il nous semble donc important à l'avenir de planifier ce type de séance plus souvent tout au long d'un projet, pour une meilleure productivité.

Pour conclure, ce projet a été une expérience enrichissante, autant du point de vue technique et de la planification, que du travail en grand groupe. La collaboration au sein de l'équipe a été très bonne durant toute la durée du semestre, et cela même malgré la grande charge de travail avec les autres cours. Nous sommes donc au final heureux d'avoir pu participer à l'élaboration de A à Z d'un projet complet et complexe, nous profitons de cette conclusion pour nous remercier les uns les autres pour le travail accompli.

Table des figures

1	Couches logicielles	6
2	Diagramme de classes de la représentation des graphes	9
3	Diagramme de classes de la fabrication des éléments graphiques	11
4	Diagramme de la classe GraphWidget	12
5	Diagramme de la classe GraphExporter	12
6	Diagramme de la classe GraphSceneManager	13
7	Proposition d'architecture pour l'interpréteur	16
8	Architecture générale de l'interpréteur	16
9	Diagramme de classes de l'interpréteur [3]	17
10	Diagramme de classe pour les graphes	21
11	Classe Edge solution 1	22
12	Classe Edge solution 2	23
13	Classe Edge solution finale	23
14	Classe Graph solution finale	24
15	Interface IGraph	25
16	Matrice d'adjacence de char	26
17	Matrice d'adjacence de bit	26
18	Matrice d'adjacence de collection d'Edge	26
19	Liste d'adjacence	27
20	Pattern Visitor pour l'application des algorithmes aux graphes	29
21	Comparaison entre les dates prévues et effectives des tâches principales	33
22	Comparaison entre le temps prévu et temps effectif des tâches principales	34
23	Flux des données dans l'interpréteur	38
24	Exemple d'arbre d'appels de traitements	49
25	Évolution d'un arbre d'appels des traitements	51

Références

- [1] Qt wiki, Qt coding style,
https://wiki.qt.io/Qt_Coding_Style
- [2] Faculty of Technology, Brno University of Technology, République tchèque,
<http://www.fit.vutbr.cz/study/courses/APR/public/ebnf.html>
- [3] PlantUML, Open-source tool that uses simple textual descriptions to draw UML diagrams,
<http://plantuml.com/>
<http://www.planttext.com/planttext>
- [4] Graphs data structures,
https://en.wikipedia.org/wiki/Adjacency_matrix#Data_structures
- [5] Boost.Spirit (v2.5.2), is an object-oriented, recursive-descent parser and output generation library for C++,
http://www.boost.org/doc/libs/1_60_0/libs/spirit/doc/html/index.html
- [6] Robert Sedgewick, Kevin Wayne.
Algorithms, fourth edition
<http://algs4.cs.princeton.edu/lectures/52Tries.pdf>
- [7] Robert Sedgewick.
Course on ternary search tries <https://www.youtube.com/watch?v=CIGyew07868>
- [8] DrMemory,
<http://drmemory.org/>

8 Annexes EGLI

8.1 Grammaire EBNF

L'analyse étant finie (section 5.1.1), on peut à présent développer les règles de production EBNF du langage. On part des règles de haut niveau pour descendre dans la hiérarchie (inspiration : [2]).

Point de départ (entrée utilisateur)

$\langle start \rangle \quad := \langle statement \rangle \{ \langle statement \rangle \}$

Déclarations (statements)

$\langle statement \rangle \quad := (\langle function-call \rangle \mid \langle declaration \rangle \mid \langle assignation \rangle) \text{ ';' }$

$\langle function-call \rangle \quad := \langle identifier \rangle \langle parameter-list \rangle$

$\langle declaration \rangle \quad := \langle type \rangle \langle identifier \rangle \text{ '=' } \langle parameter \rangle$

$\langle assignation \rangle \quad := \langle variable \rangle \text{ '=' } \langle parameter \rangle$

$\langle parameter-list \rangle \quad := \text{'(' } \langle parameter \rangle \{ \text{' , ' } \langle parameter \rangle \} \text{')' }$

$\langle parameter \rangle \quad := \langle constant \rangle \mid \langle indexed-array \rangle \mid \langle function-call \rangle$

Expressions et opérations

$\langle indexed-array \rangle \quad := \langle variable \rangle \text{' [' } \langle digit-sequence \rangle \text{']' }$

Types de variable et d'enregistrement

$\langle type \rangle \quad := \langle simple-type \rangle \mid \langle complex-type \rangle$

$\langle simple-type \rangle \quad := \text{'Boolean' } \mid \text{'Number' } \mid \text{'Integer' } \mid \text{'Float' } \mid \text{'String' }$

$\langle complex-type \rangle \quad := \text{'Array' } \mid \text{'Graph' } \mid \text{'Vertex' } \mid \text{'Edge' }$

$\langle array-record \rangle \quad := \text{' [' } \langle constant \rangle \{ \text{' , ' } \langle constant \rangle \} \text{']' }$

$\langle constant \rangle \quad := \langle simple-constant \rangle \mid \langle complex-constant \rangle$

$\langle complex-constant \rangle \quad := \langle array-record \rangle \mid \langle graph-record \rangle \mid \langle edge-record \rangle \mid \langle vertex-record \rangle$

$\langle graph-record \rangle \quad := \text{'{' } \langle graph-info \rangle \{ \text{' , ' } \langle graph-info \rangle \} \text{' } \mid \text{'{' } \# \langle digit-sequence \rangle \{ \text{' , ' } \langle graph-info \rangle \} \text{' } }$

$\langle edge-record \rangle \quad := \text{'(' } \langle edge-info \rangle \text{')' }$

$\langle vertex-record \rangle \quad := \text{'(' } \langle vertex-info \rangle \text{')' }$

$\langle graph-info \rangle \quad := \langle vertex-info \rangle \mid \langle edge-info \rangle$

$\langle edge-info \rangle \quad := \langle connection \rangle \mid \langle connection \rangle \text{' : ' } \langle number \rangle \mid \langle connection \rangle \text{' : ' } [\langle number \rangle] \text{' : ' } \langle string \rangle \mid \langle connection \rangle \text{' : ' } [\langle number \rangle] \text{' : ' } [\langle string \rangle] \text{' : ' } \langle number \rangle \mid \langle connection \rangle \text{' : ' } [\langle number \rangle] \text{' : ' } [\langle string \rangle] \text{' : ' } [\langle number \rangle] \text{' : ' } \langle number \rangle$

$\langle vertex-info \rangle \quad := \langle id \rangle \mid \langle id \rangle \text{' : ' } \langle string \rangle \mid \langle id \rangle \text{' : ' } [\langle string \rangle] \text{' : ' } \langle number \rangle \mid \langle id \rangle \text{' : ' } [\langle string \rangle] \text{' : ' } [\langle number \rangle] \text{' : ' } \langle number \rangle \mid \langle id \rangle \text{' : ' } [\langle string \rangle] \text{' : ' } [\langle number \rangle] \text{' : ' } [\langle number \rangle] \text{' : ' } \langle number \rangle$

$\langle connection \rangle \quad := \langle id \rangle (\text{' -- ' } \mid \text{' -> ' } \mid \text{' <- ' }) \langle id \rangle$

$\langle id \rangle \quad := \langle digit-sequence \rangle$

Identifiants et définitions de bas niveau

$\langle simple-constant \rangle \quad := \langle sign \rangle \langle variable \rangle \mid \langle number \rangle \mid \langle boolean \rangle \mid \langle string \rangle$

$\langle variable \rangle \quad := \langle identifier \rangle$

$\langle identifier \rangle \quad := \langle letter \rangle \{ \langle letter \rangle \mid \langle digit \rangle \mid \text{' _ ' } \}$

$\langle string \rangle \quad := \text{' ' } \{ \langle letter \rangle \mid \text{' ' } \} \text{' ' }$

$\langle letter \rangle \quad := \text{? all lower and upper case letters?}$

$\langle \text{boolean} \rangle$	$:= \text{'True'} \mid \text{'False'}$
$\langle \text{number} \rangle$	$:= \langle \text{integer-number} \rangle \mid \langle \text{real-number} \rangle$
$\langle \text{real-number} \rangle$	$:= [\langle \text{sign} \rangle] \langle \text{digit-sequence} \rangle \text{'.'} \langle \text{digit-sequence} \rangle$
$\langle \text{integer-number} \rangle$	$:= [\langle \text{sign} \rangle] \langle \text{digit-sequence} \rangle$
$\langle \text{digit-sequence} \rangle$	$:= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \}$
$\langle \text{sign} \rangle$	$:= \text{'+'} \mid \text{'-'}$
$\langle \text{digit} \rangle$	$:= ? \text{ all digits from 0 to 9}$

8.1.1 Modifications

24.04.2016

Après réflexion, il est apparu que les opérations d'assignation et de déclaration de variables pouvaient être mises ensemble. Concrètement dans une déclaration il n'est pas nécessaire de définir le type de la variable car celui-ci peut être déduit de l'opérande de droite. Et comme les tableaux sont des collections hétérogènes, il semble évident que les variables doivent elles aussi être à typage dynamique. Par conséquent la grammaire est modifiée ainsi :

- Suppression des règles $\langle \text{declaration} \rangle$, $\langle \text{type} \rangle$, $\langle \text{simple-type} \rangle$ et $\langle \text{complex-type} \rangle$
- La règle $\langle \text{assignment} \rangle$ vaut désormais : $(\langle \text{variable} \rangle \mid \langle \text{identifier} \rangle) \text{'='} \langle \text{parameter} \rangle$
- La règle $\langle \text{statement} \rangle$ vaut désormais : $(\langle \text{function-call} \rangle \mid \langle \text{assignment} \rangle) \text{';'}$

12.05.2016

Pendant l'implémentation du parseur, des oublis dans la grammaire ont été constatés, voici les règles manquantes :

- Nouvelle règle : $\langle \text{graph-add} \rangle : := \langle \text{identifier} \rangle \text{'+'} \langle \text{parameter} \rangle$
- Nouvelle règle : $\langle \text{graph-sub} \rangle : := \langle \text{identifier} \rangle \text{'-'}$ $\langle \text{parameter} \rangle$
- La règle $\langle \text{connection} \rangle$ vaut désormais : $\langle \text{id} \rangle (\text{'—'} \mid \text{'>'} \mid \text{'<'}) \langle \text{id} \rangle [\text{'['} \langle \text{digit-sequence} \rangle \text{']'}]$
- La règle $\langle \text{statement} \rangle$ vaut désormais : $(\langle \text{function-call} \rangle \mid \langle \text{assignment} \rangle \mid \langle \text{graph-add} \rangle \mid \langle \text{graph-sub} \rangle) \text{';'}$

De plus, en l'état, l'implémentation ne permet pas d'écrire par exemple $v = (0:\text{label})$; ou $v = (0::-2)$;, cela à cause des règles qui n'utilisent pas assez la règle $\langle \text{parameter} \rangle$. Cela peut être une amélioration future.

8.2 Flux des données

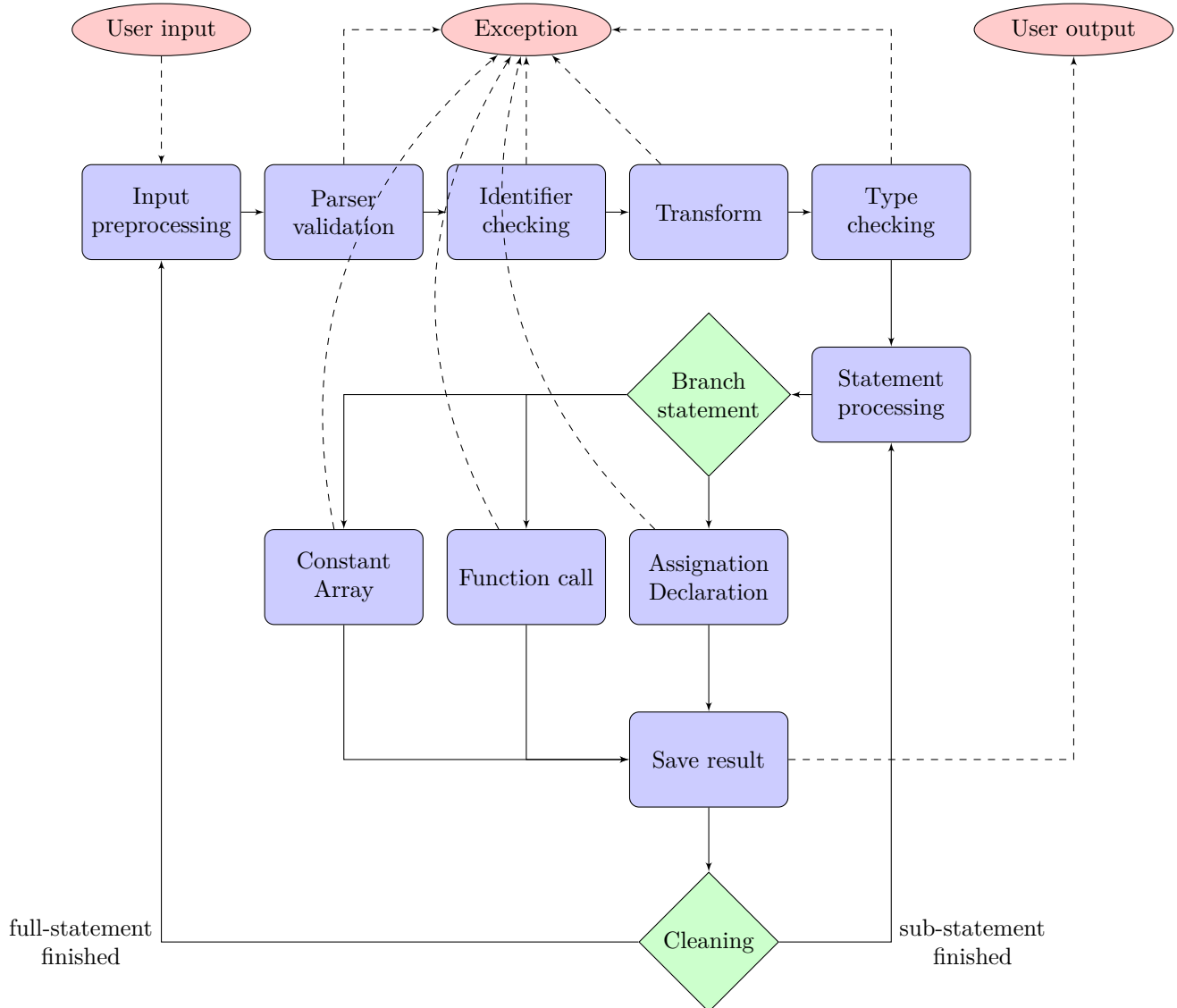


FIGURE 23 – Flux des données dans l'interpréteur

Détaillons chacune de ces étapes :

1. **Input preprocessing** - On récupère les entrées utilisateur dans un tampon.
 - (a) **Trim** - On enlève tous les espaces inutiles, c'est-à-dire tous les espaces sauf ceux dans les constantes littérales de type *String*, et celui qui sépare le type de l'identifiant dans les déclarations.
 - (b) **Statement per statement** - On envoie au parseur une commande à la fois (les traitements sont séparés par des ';').
2. **Parser validation** - On vérifie la syntaxe de la commande et on extrait des informations utiles pour la suite (tokens), voir section 8.3.
3. **Identifier checking** - On connaît les différents identifiants présents dans la commande (noms de variable, noms de fonction), on peut tout de suite vérifier leur existence.
4. **Transform** - On transforme la commande en arbre d'appels, plus de détails dans la section 8.6.
5. **Type checking** - On vérifie la concordance des types (statique), plus de détails dans la section 8.6.4.
6. **Statement processing** - La commande a été décomposée en plusieurs sous-traitements (arbre d'appels, 8.6.3), on effectue d'abord les feuilles avant de remonter.
 - (a) **Branch statement** - On sélectionne le bon traitement en fonction de son type.

- **Constant** - On crée une variable temporaire avec la valeur de la constante, voir section 8.4.2.
- **Array** - On crée un tableau dynamique hétérogène, voir section 8.4.3.
- **Function call** - On appelle la fonction et on sauve son résultat dans une variable temporaire.
- **Assignment** - On assigne une valeur à une variable, voir section 8.4.1.
- **Declaration** - Pareil que l'assignation, mais avec création de l'identifiant au préalable.
- (b) **Save result** - On transforme l'arbre d'appels avec les nouvelles valeurs/variables. Si c'est la fin de la commande, on indique à l'utilisateur que le résultat de la commande est prêt
- (c) **Cleaning** - On supprime les variables temporaires qui ne sont plus utiles. Si c'était un sous-traitement, on passe au prochain sous-traitement (6), sinon on passe à la prochaine commande (1)

8.3 Implémentation du parseur

Le parseur a deux but principaux :

- vérifier la validité syntaxique de l'entrée utilisateur
 - ... et indiquer clairement où se trouve l'erreur si il y en a une
- générer des *tokens* décrivant l'entrée utilisateur
 - ... utilisables dans la suite du flux de données

L'implémentation ad-hoc d'un tel parseur est une tâche compliquée, ainsi nous avons décider d'utiliser la bibliothèque Boost.Spirit [5] qui va nous permettre d'accélérer grandement le temps de développement de cette partie.

Nous n'allons pas entrer ici dans les détails d'utilisation de Boost.Spirit, les tutoriels en ligne sont suffisamment bien écrits. Notons simplement que cette bibliothèque permet de construire rapidement un parseur depuis une grammaire EBNF, et permet de constituer les *tokens* tout aussi rapidement.

NB : les *tokens* générés par le parseur sont décrits dans la section 8.6.

8.4 Mémoire virtuelle

Dans le flux de données, certaines étapes nécessitent l'utilisation d'une mémoire afin de gérer les variables (temporaires ou non). Dans cette section nous allons préciser comment cela va être géré.

Chaque type du langage possède son pendant en C++, voici un tableau résumant cela :

Type	Equivalent C++
Boolean	bool
Integer	int
Float	float
String	std::string
Number	voir section 8.4.4
Array	voir section 8.4.3
Graph	voir section 6
Vertex	voir section 8.4.4
Edge	voir section 8.4.4

8.4.1 Table des variables

Notre interpréteur a besoin de créer des variables afin de pouvoir les manipuler au fur et à mesure des traitement. Une variable est définie par trois informations :

1. un type
2. un identifiant
3. une valeur

Pour stocker ces informations, on va utiliser une table de variables qui nous permettra de répondre à différentes questions :

- Quelles sont les variables déjà créées ?
- Quel est le type d'une variable ?
- Une variable existe-t-elle ?
- Quelle est la valeur d'une variable ?

Le C++ étant un langage fortement typé, on est obligé d'utiliser une approche à base de `if-else/switch`. Voici un exemple d'implémentation (qui pourrait être *templatisé*) :

```
struct Table {
    enum class Type { Integer, String /* , ... */ };
    map<string, Type> names;
    map<string, int> ints;
    map<string, string> strings;
    // ...

    void set(string const &name, int value) {
        if(exists(name) && typeOf(name) != Type::Integer)
            /* bad type exception */;
        names[name] = Type::Integer;
        ints[name] = value;
    }

    void set(string const &name, string const &value) {
        // ...
    }
    // ...

    bool exists(string const &name) const {
        return names.find(name) != names.end();
    }

    Type typeOf(string const &name) const {
        if(!exists(name))
            /* not found exception */;
        return names.find(name)->second;
    }

    void erase(string const &name) {
        switch(typeOf(name)) {
            case Type::Integer: ints.erase(name); break;
            case Type::String: strings.erase(name); break;
        }
    }

    int getInteger(string const &name) const {
        if(typeOf(name) != Type::Integer)
            /* cast exception */;
        return ints.find(name)->second;
    }
    // ...
};
```

Cette solution répond à toutes nos questions et a l'avantage d'être assez simple (et typée), une autre approche à base de `void*` est possible et est présentée dans la section 8.4.3 sur les tableaux dynamiques hétérogènes. Notons qu'il peut être pertinent d'utiliser une autre structure que `map` pour le stockage des identifiants, par exemple un ternary search tries qui faciliterait l'auto-complétion pour les variables.

8.4.2 Table des temporaires

Certains traitements seront composés de plusieurs sous-traitements, les résultats de ceux-ci seront stockés dans des variables qui doivent avoir une durée de vie limitée à la durée du traitement. Une approche simple et efficace est d'utiliser une `std::stack<Table>` (voir section 8.4.1) sur laquelle on *push* à l'entrée du traitement

et on *pop* à sa sortie.

Ici nous développons juste l'idée générale, mais concrètement les variables temporaires et non temporaires doivent être accessible de la même manière indépendamment de leur genre. Par conséquent une tables des variables générale englobera les deux genres, et fournira une interface commune permettant de manipuler les deux.

8.4.3 Tableau dynamique hétérogène

Le tableau dynamique hétérogène (TDH dans la suite de cette section) pose un problème majeur : le C++ est un langage fortement typé. Cependant plusieurs solutions s'offrent à nous pour créer un TDH :

1. Un `std::vector<boost::any>` (`std::any` étant prévu pour C++17)
2. Un `std::vector<std::pair<std::type_index, void*> >`
3. Un `std::vector<std::pair<TypeEnum, void*> >`

La solution (1) est sans doute la plus propre ainsi que la plus facile, cependant elle ajoute une dépendance avec Boost et elle partage plusieurs problèmes avec la solution (2) que l'on va voir tout de suite.

La solution (2) utilise un mécanisme de C++ (l'opérateur `typeid`) afin de savoir quel est le type réel caché derrière le `void*`, cela permet un code assez générique. Cependant pour chaque valeur, on doit attacher un objet `std::type_index` (lui-même composé d'un `std::type_info`), et faire des appels à `typeid`, cela ajoute un surcoût en temps et en mémoire par rapport à la solution (3).

La solution (3) est donc celle retenue pour implémenter un TDH. On crée simplement une énumération contenant les types acceptés dans le TDH, et chaque `void*` est associé à une valeur de cette énumération. Voici un exemple d'implémentation :

```
class Array {
public:
    enum class Type { Boolean, Integer, String /*, ...*/ };
    typedef std::pair<Type, void*> value_type;

    // constructeur de copie, etc.
    // ...

    ~Array() {
        for(value_type& value : values) {
            switch(value.first) {
                case Type::Boolean:
                    delete (bool*)value.second;
                    break;
                case Type::Integer:
                    delete (int*)value.second;
                    break;
                case Type::String:
                    delete (std::string*)value.second;
                    break;
                // ...
            }
        }
    }

    size_t size() const {
        return values.size();
    }

    void add(int *value) {
        // value doit provenir d'un 'new'
        values.push_back(std::make_pair(Type::Integer, (void*)value));
    }

    void add(int value) {
        add(new int(value));
    }
}
```

```
// ...

Type typeOf(size_t i) const {
    return values.at(i).first;
}

int getInteger(size_t i) const {
    if(typeOf(i) != Type::Integer)
        throw /* cast exception */;
    return *(int*)values.at(i).second;
}
// ...

private:

    std::vector<value_type> values;
};
```

Cette solution est assez verbeuse, cependant la complexité en temps et en mémoire est très bonne. En ce qui concerne les `delete`, il est obligatoire de faire le cast avant, sinon le destructeur de l'objet détruit n'est pas appelé.

Évidemment l'utilisateur du TDH est obligé de faire des `switch/if-else` dans son code, cependant ce problème vaut pour chacune des trois solutions et est inhérent au C++.

8.4.4 Number, Edge et Vertex

Number - Selon la grammaire, certains nombres peuvent soit être un entier, soit un flottant. Il peut également être intéressant pour certains algorithmes de savoir si ces nombres sont entiers ou non (pour les flots notamment). Pour ces deux raisons, il est nécessaire d'avoir un type `Number` pouvant stocker les deux types de représentations. Voici deux propositions (incomplètes) :

```
// solution 1
struct Number {
    enum : char { Integer, Float } tag;
    union { int int_; float float_; };
    Number() : Number(0) {}
    Number(int i) : tag(Integer), int_(i) {}
    Number(float f) : tag(Float), float_(f) {}
    Number(double f) : tag(Float), float_(f) {}
    // ...
};

// solution 2
struct Number2 {
    typedef float T;
    T value;
    Number2(T v = T(0)) : value(v) {}
    operator T() const { return value; }
    operator T&() { return value; }
    explicit operator int() const { return value; }
    bool isInt() const { return modf(value, nullptr) == 0.f; } // +/- epsilon
    bool isFloat() const { return !isInt(); }
    // ...
};
```

La solution (1) a l'avantage d'être plus sûre au niveau du typage, cependant à l'utilisation elle demande davantage de conditions. La solution (2) est plus naturelle à l'utilisation et utilise deux fois moins de place en mémoire que la solution (1), mais le test `isInt()` prend plus de temps. A partir de là, on a choisi la solution (2) pour ses avantages et car les accès à `value` seront plus courants que les tests `isInt()`.

Vertex - Le type `Vertex` pose un problème majeur : certains de ses attributs sont optionnels. Voici donc une solution naturelle :

```
struct Vertex {
    int id;
    std::string label;
```

```
    Number weight;  
    Number minCapacity;  
    Number maxCapacity;  
    bool active[4]; // ~ un bool par attribut optionnel  
    // ...  
};
```

Le souci avec cette approche est que les attributs optionnels sont construits même s'ils ne sont pas utilisés, par ailleurs sur mon ordinateur `sizeof(Vertex) == 36`. Cela implique une utilisation inutile en ressources pour des attributs qui ne seront pas forcément utilisés.

Afin d'éviter ces problèmes, on peut approcher le problème avec des pointeurs (égaux à `nullptr` s'ils ne sont pas utilisés) :

```
struct Vertex {  
    int id;  
    std::string* label;  
    Number* weight;  
    Number* minCapacity;  
    Number* maxCapacity;  
    // ...  
};
```

A présent, `sizeof(Vertex) == 20` (sur ma machine) et les objets ne sont plus construits inutilement. Le gain en ressource n'est pas négligeable, cependant cette structure n'est pas très agréable à utiliser telle quelle (gestion de l'allocation/désallocation, déréférencement). Pour palier à cela, on délègue la gestion du pointeur à une autre classe, dont voici un aperçu :

```
template<typename T>  
struct Optional {  
    T* value;  
  
    // constructeur/destructeur  
    Optional() : value(nullptr) {}  
    ~Optional() { delete value; }  
  
    // operations de copie  
    Optional(const Optional &o) : value(nullptr) {  
        if(o.value) value = new T(*o.value);  
    }  
    Optional(Optional &&o) : value(move(o.value)) {}  
    Optional &operator=(const Optional &o) {  
        if(this != &o) {  
            delete value;  
            value = nullptr;  
            if(o.value) value = new T(*o.value);  
        }  
        return *this;  
    }  
    Optional &operator=(Optional &&o) {  
        if(this != &o) {  
            delete value;  
            value = move(o.value);  
        }  
        return *this;  
    }  
  
    // affectations et constructions  
    Optional(const T& v) { value = new T(v); }  
    Optional(T&& v) : value(new T(move(v))) {}  
    Optional &operator=(const T &v) {  
        if(value) *value = v;  
        else value = new T(v);  
        return *this;  
    }  
    Optional &operator=(T &&v) {  
        if(value) *value = move(v);  
    }  
};
```

```
        else value = new T(move(v));
        return *this;
    }

    // ...
    // opérateurs d'accès, etc.
    // ...
};

struct Vertex {
    int id;
    Optional<string> label;
    Optional<Number> weight;
    Optional<Number> minCapacity;
    Optional<Number> maxCapacity;
    // ...
};
```

Notre type `Vertex` est à présent élégant, et plus efficace en terme de ressources. Notons que `std::optional<>` sera disponible en C++17.

Edge - En ce qui concerne le type `Edge`, on peut utiliser la même démarche que pour le type `Vertex` :

```
struct Edge {
enum Connection : char { Unidirectional, Bidirectional };
    int a, b;
    Connection conn;
    Optional<Number> weight;
    Optional<string> label;
    Optional<Number> minCapacity;
    Optional<Number> maxCapacity;
    // ...
};
```

Nos types "simples" sont maintenant définis, notons que les exemples d'implémentation ci-dessus ne sont pas complets, mais ils montrent l'approche générale.

8.5 Table des fonctions

Dans le flux de données, l'étape *Function call* nécessite d'appeler, d'exécuter et de récupérer le résultat d'une fonction. Dans cette section nous allons préciser comment cela va être fait.

Voici l'idée avec un bout de code qui devrait pouvoir être exécuté :

```
// algorithme independant de l'interpreteur
Graph dijkstra(const Graph &graph, const Vertex &origin) {
    // ...
}

// interfacage dans la table des fonctions
functions.add("dijkstra", &dijkstra);

// appel a la fonction depuis l'interpreteur
functions.call("dijkstra", "pcc", {"g", "v1"});
```

Ce `call` va chercher les variables `g` et `v1` dans la table des variables, puis appelle la fonction `dijkstra` avant de sauver le résultat dans la table des variables sous le nom `pcc`.

La difficulté, comme d'habitude, est de lier le côté dynamique de l'interpréteur avec le côté fortement typé de C++. Le deuxième problème est d'avoir un interfaçage facile à faire côté utilisateur, afin que l'ajout de nouvelle fonction soit rapide et élégant.

Une liste des fonctions interfacées dans l'application se trouve dans l'annexe 5.3.1.

8.5.1 Interfaçage

Afin de répondre aux demandes de la table des fonctions (voir section 8.5), on va utiliser deux outils que le C++ nous offre : l'héritage (avec polymorphisme) et la spécialisation des `template`.

Un exemple d'implémentation vaut plus qu'un long discours :

```
// Types geres
enum class Type { Integer, String /* , ... */ };

// Table des variables (simplifiée)
struct VarTable {
    map<string, Type> names;
    map<string, int> ints;
    map<string, string> strings;

    void set(string const &name, int value) {
        names[name] = Type::Integer;
        ints[name] = value;
    }

    void set(string const &name, string const &value) {
        names[name] = Type::String;
        strings[name] = value;
    }
};

// Accés aux variables specialise
template<typename> struct Fetcher;

template<> struct Fetcher<int> {
    static int get(string const &name, VarTable &var) {
        return var.ints[name];
    }
};

template<> struct Fetcher<string> {
    static string const &get(string const &name, VarTable &var) {
        return var.strings[name];
    }
};

// Enleve les 'const' et les '&' sur un type
template<typename T> struct RemoveAll {
    typedef typename remove_const<typename remove_reference<T>::type>::type type;
};

// Classe abstraite pour l'appel a une fonction
struct FuncBase {
    virtual ~FuncBase() {}
    virtual void execute(string const&, vector<string> const&, VarTable&) = 0;
};

// Type inutile pour les specialisations de la classe Func
struct Empty {};

// Classe appelant notre fonction
// (ici avec 2 parametres max, mais peut etre etendu)
template<typename R, typename P1 = Empty, typename P2 = Empty>
struct Func : FuncBase {
    function<R(P1, P2)> f;
    Func(R(*f)(P1, P2)) : f(f) {}
    virtual ~Func() {}

    // on appelle la fonction avec ses parametres
    void execute(string const &dst, vector<string> const &params, VarTable &var) {
        P1 p1 = Fetcher<typename RemoveAll<P1>::type>::get(params[0], var);
```

```

    P2 p2 = Fetcher<typename RemoveAll<P2>::type>::get(params[1], var);
    var.set(dst, f(p1, p2));
}
};

// Meme chose mais pour 1 parametre
template<typename R, typename P1>
struct Func<R, P1, Empty> : FuncBase {
    function<R(P1)> f;
    Func(R(*f)(P1)) : f(f) {}
    virtual ~Func() {}
    void execute(string const &dst, vector<string> const &params, VarTable &var) {
        P1 p1 = Fetcher<typename RemoveAll<P1>::type>::get(params[0], var);
        var.set(dst, f(p1));
    }
};

// Meme chose pour 0 parametre
template<typename R>
struct Func<R, Empty, Empty> : FuncBase {
    function<R()> f;
    Func(R(*f)()) : f(f) {}
    virtual ~Func() {}
    void execute(string const &dst, vector<string> const &params, VarTable &var) {
        var.set(dst, f());
    }
};

// On rend la creation de Func simple pour l'utilisateur
template<typename R, typename P1, typename P2>
FuncBase *makeFunc(R(*f)(P1, P2)) {
    return new Func<R, P1, P2>(f);
}
template<typename R, typename P1>
FuncBase *makeFunc(R(*f)(P1)) {
    return new Func<R, P1>(f);
}
template<typename R>
FuncBase *makeFunc(R(*f)()) {
    return new Func<R>(f);
}

// Table des fonctions (simplifiee)
struct FuncTable {
    VarTable &var;
    map<string, FuncBase*> names;

    FuncTable(VarTable &var) : var(var) {}
    ~FuncTable() {
        for(auto p : names)
            delete p.second;
    }

    // on appelle la fonction 'name'
    void execute(string const &name, string const &dst, vector<string> const &params) {
        // verifications eventuelles
        // ...

        names[name]->execute(dst, p, var);
    }
};

// Fonctions de test
int add(int a, int b) { return a + b; }
int inv(int a) { return -a; }
int len(string const &s) { return s.size(); }

```

```
string hw() { return "helloworld"; }

int main()
{
    // On cree des variables
    VarTable var;
    var.names["a"] = Type::Integer;
    var.ints["a"] = 2;
    var.names["b"] = Type::Integer;
    var.ints["b"] = 3;

    // On interface les fonctions
    FuncTable func(var);
    func.names["add"] = makeFunc(&add);
    func.names["inv"] = makeFunc(&inv);
    func.names["len"] = makeFunc(&len);
    func.names["hw"] = makeFunc(&hw);

    // Affichera: -5
    func.execute("add", "c", {"a", "b"});
    func.execute("inv", "c", {"c"});
    cout << var.ints["c"] << endl;

    // Affichera: helloworld/10
    func.execute("hw", "s", {});
    func.execute("len", "l", {"s"});
    cout << var.strings["s"] << "/" << var.ints["l"] << endl;

    return 0;
}
```

Le coeur de la solution est la classe de base abstraite **FuncBase** dont hérite la classe *templatisée* **Func** qui effectue l'appel effectif à la fonction interfacée. On remarque que cela fonctionne bien, et que côté utilisateur l'ajout d'une nouvelle fonction est très agréable. Il est de plus encore possible d'ajouter des informations (nom, arité, types, ...) sur chacune des fonctions interfacées.

Remarque : L'implémentation ci-dessus utilise la spécialisation des *template* et est limitée en terme de nombre de paramètres que les fonctions interfacées peuvent avoir (même si c'est facilement extensible). Une autre approche, plus générique encore, est d'utiliser les *variadic template* de C++11, cependant cette solution est plus compliquée à développer, le choix est donc laissé au développeur.

8.5.2 Surcharge

Nous avons vu comment interfacier des fonctions simplement, cependant il n'est pour l'instant pas possible d'interfacier plusieurs fonctions ayant le même nom mais une signature différente. Nous allons voir à présent comment cela va être géré.

Il existe sans doute plusieurs manières de faire cela, une solution simple consiste à :

- modifier la **map** dans **FuncTable** en une **multimap**
- ajouter une méthode abstraite `bool match(vector<string> const ¶ms, VarTable &var)` dans **FuncBase**
- modifier la méthode **execute** de **FuncTable** afin qu'elle exécute la première **FuncBase** qui **match** les paramètres passés

Un test de faisabilité a été fait et cela marche très bien, on ne mets pas le code entier ici car c'est le même que précédemment. Cependant la méthode **match** est quand même intéressante :

```
// ...

template<typename T> struct TypeOf;
template<typename T> struct TypeOf<T&> : TypeOf<T> {};
template<typename T> struct TypeOf<const T> : TypeOf<T> {};
template<> struct TypeOf<int> { static constexpr Type value = Type::Integer; };
template<> struct TypeOf<string> { static constexpr Type value = Type::String; };

// ...
```

```
// dans la classe Func a deux parametres (heritant de FuncBase)
virtual bool match(vector<string> const& params, VarTable& var) {
    return params.size() == 2
        && TypeOf<P1>::value == var.names[params[0]]
        && TypeOf<P2>::value == var.names[params[1]];
}

// ...
```

Une autre solution serait d'utiliser une `map<pair<string, vector<Type> >, FuncBase*>` et de construire la signature (`vector<Type>`) lors de l'interfaçage et lors de l'appel à une fonction.

8.6 Règles de transformation

Dans cette section nous allons préciser les règles de transformation de l'étape *Transform* du flux de données, ainsi que ce que cela va apporter. Pour commencer, voici quelques exemples de transformations possibles :

```
(1::3)    → _vertex_create(1,-,3)
(0->1:2::3) → _edge_create_unidirectional(0,1,2,-,3)
[1,2,3]    → _array_add(_array_add(_array_add(_array_create(),1),2),3)
{#3,0->1,1->2:6} → {1,2,3,0->1,1->2:6}
              → {[ (1), (2), (3), (0->1), (1->2:6) ]}
              → _graph_create(_array_add(...))
g+=(3)     → g=_graph_add(g,_vertex_create(3,-,-,-,-))
```

Comme on le voit, le but est de passer de la syntaxe du langage à une représentation sous forme de fonctions. Plus précisément sous une forme d'arbre représentant la hiérarchie d'appel des différents traitements et sous-traitements.

On va prendre un exemple de départ complet, et on va développer l'arbre correspondant. Voici notre exemple qui contient tout ce qui nous intéresse :

```
pcc=dijkstra({v0,1:Yverdon,e01,2,0->2:9,1<-2:3},v0) (avec v0=(0) et e01=(0->1))
```

- assignation (avec déclaration)
- appel de fonction
- identifiant passé en paramètre
- temporaires passés en paramètre
- constantes littérales

Et voici l'arbre correspondant :

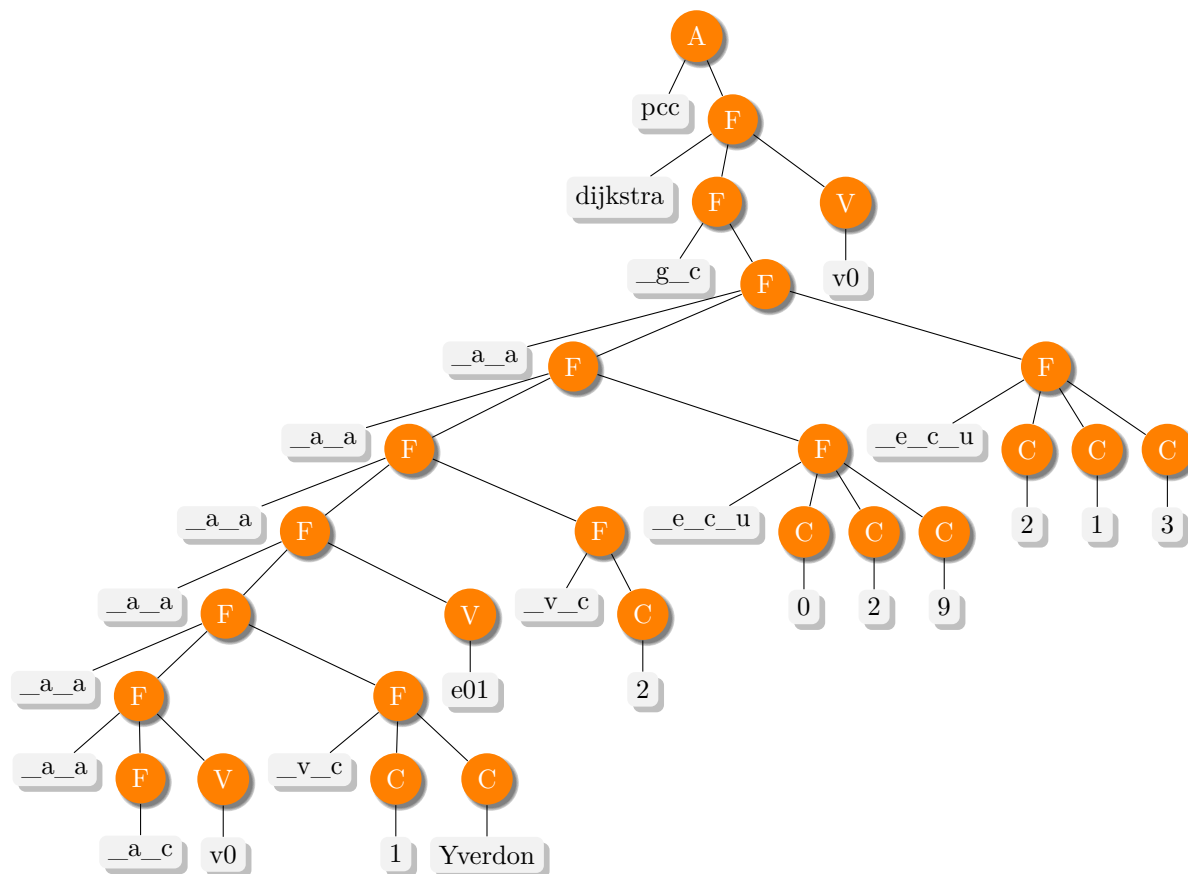


FIGURE 24 – Exemple d'arbre d'appels de traitements

NB : on a simplifié les noms des fonctions afin que l'image ne soit pas trop large (`_g_c=_graph_create`, `_a_a=_array_add`, `_a_c=_array_create`, `_v_c=_vertex_create` et `_e_c_u=_edge_create_unidirectional`).

Dans ce graphe on a deux types de noeuds, les ronds qui représentent des traitements, et les rectangles qui représentent des valeurs. On remarque qu'on a plusieurs types de traitements :

Type	Nom complet	Valeur	Nombre de paramètres
A	Assignation	Nom de la variable	1
F	Fonction	Nom de la fonction	*
C	Constante	Valeur littérale	0
V	Variable	Nom de la variable	0

D'un point de vue structure de données cet arbre est très simple, voici un exemple :

```
struct Statement {
    enum class Type { A, F, C, V };
    Type type;
    string value;
    vector<Statement> parameters;
};
```

La création de cet arbre se fait grâce aux *tokens* générés dans la partie *Parser validation* (voir section 8.3).

8.6.1 Modifications

24.04.2016

Suite à l'implémentation du parseur, il est apparu qu'un nouveau type de *Statement* était nécessaire afin de simplifier certains développement. Il s'agit du type Array, voici les types de traitements remagnés :

Type	Nom complet	Valeur	Nombre de paramètres
A	Assignation	Nom de la variable	1
F	Fonction	Nom de la fonction	*
C	Constante	Valeur littérale	0
V	Variable	Nom de la variable	0
Ac	Création d'un tableau	<vide>	*
Ai	Accès indexé dans un tableau	Nom de la variable	1

8.6.2 Fonctions internes

La transformation nécessite plusieurs fonctions pour que la suite puisse correctement fonctionner. Nous allons lister ici toutes ces fonctions qui seront par la suite interfacées dans la table des fonctions.

Note : les types de retour ainsi que les paramètres sont définis par l'implémentation.

Prototype	Description
<code>--edge_create</code>	Création d'un Edge, ex : <code>e = (0->1:3::5);</code>
<code>--graph_add</code>	Ajout/Modif. sur un Graph, ex : <code>g += [v5, e45];</code>
<code>--graph_create</code>	Création d'un Graph, ex : <code>g = {#2, 0-1};</code>
<code>--graph_sub</code>	Suppression dans un Graph, ex : <code>g -= [v0, v1];</code>
<code>--negate</code>	Négation d'un Number, ex : <code>n = -n;</code>
<code>--vertex_create</code>	Création d'un Vertex, ex : <code>v = (0:"Yverdon");</code>

8.6.3 Arbre d'appels des traitements

Maintenant que nous avons un arbre d'appels hiérarchique des traitements, reste à savoir comment nous allons l'utiliser (étapes *Statement processing* et *Save result* du flux de données). Le principe est très simple :

1. On prend le traitement racine
2. On parcourt les paramètres :
 - Si c'est un V, on passe
 - Sinon on effectue le sous-traitement (récursion)
3. On effectue le traitement racine

Concrètement on voit que tous les traitements ont besoin d'accéder aux valeurs de leur paramètres, cela ne peut se faire que si le paramètre est une variable. Donc si ce n'est pas une variable, on effectue le traitement associé avant de le sauver dans une variable (temporaire). L'arbre va donc raccourcir au fur et à mesure de l'avancement des sous-traitements.

Prenons un exemple simple `a=1+max(-b,2)` (avec `b=-3`), qui devient donc `a=add(1,max(neg(b),2))`. L'arbre va évoluer ainsi (avec en orange le paramètre à traiter, en rouge sa variable résultante et en jaune la récursion) :

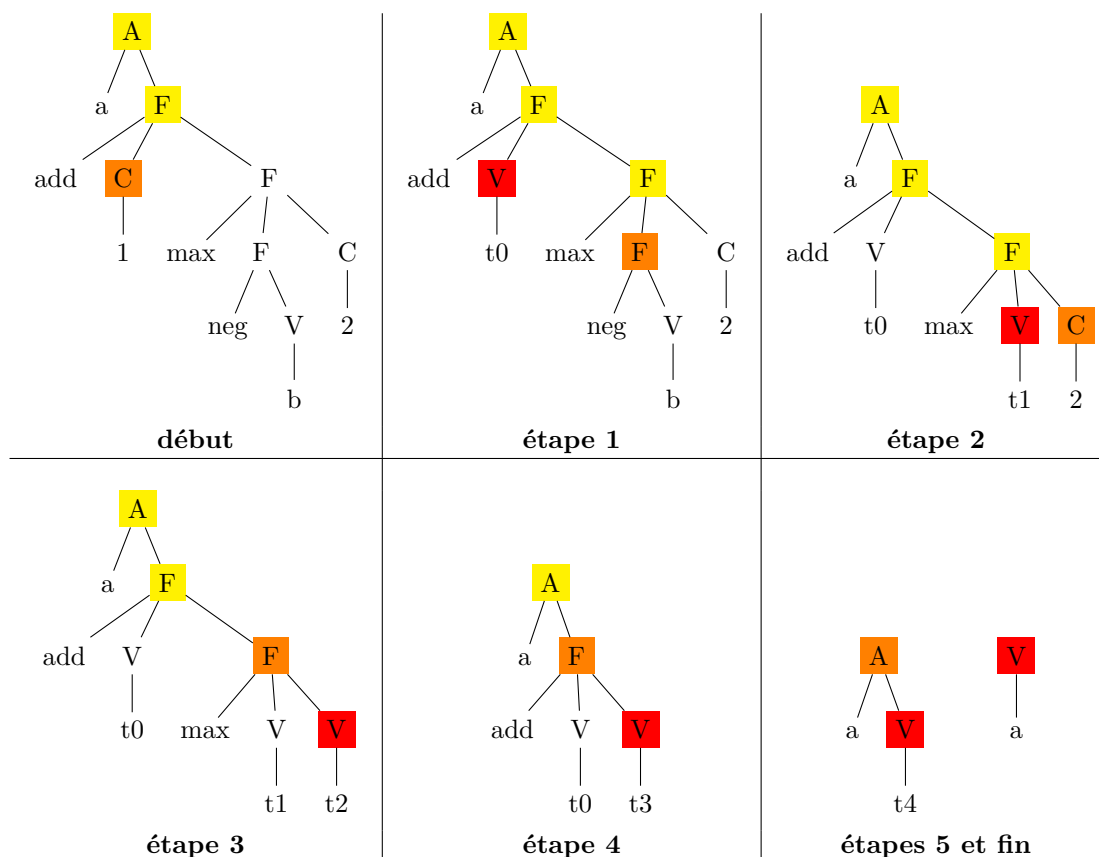


FIGURE 25 – Évolution d'un arbre d'appels des traitements

NB : les variables tX sont des variables temporaires, elles seront supprimées à la fin du (sous-)traitement qui en avait besoin.

Premièrement on voit que cette approche récursive nous permet de bien arriver au résultat escompté, et ce, depuis un traitement qui à la base était très complexe avant sa transformation. Deuxièmement il est important de noter que tout traitement finira toujours par retourner une valeur dans une variable (temporaire ou pas).

8.6.4 Vérification des types

L'arbre d'appel des traitements nous permet d'implémenter également la partie *Type checking* du flux de données. Le but de cette partie est de faire une première vérification sur la concordance des types entre les fonctions et leurs paramètres. Ainsi on évite d'effectuer pleins de sous-traitements pour finalement tout annuler car il y a un problème ailleurs dans l'arbre, plus un soucis est détecté tôt, mieux c'est.

Cela est possible car les tables des variables et des fonctions nous permet d'obtenir des informations sur leurs contenus. L'algorithme consiste à nouveau en une simple récursion dans l'arbre, et à chaque noeud on vérifie les types.

NB : cette vérification ne verra pas tous les problèmes de types, typiquement à cause des tableaux dynamiques hétérogènes.

9 Annexes projet

9.1 Cahier des charges

ANNEXE : Cahier des charges

1 Spécification

L'objectif global de ce projet est la réalisation d'une application de résolution de problèmes classiques des graphes dans le cadre du cours PRO. Elle sera réalisée en Qt afin de pouvoir être déployée sur les plateformes Linux, Mac OS et Windows mais ne sera que testée sur cette dernière.

1.1 Maquette (pas définitive)

```
In{1} << myGraph = (1, 2, 3, 4, 5, a:1->2, b:2->3, c:2->4, d:3->5, e:4->5, f:4->2, g:1->5)
In{2} << draw(myGraph)
Out{2} >>
executed in 0.00008 [ms]
In{3} << myGraph.ponderate(a:5, b:6, c:2, d:3, 4->5:4, 4->2:1, 1->5:3)
In{4} << draw(myGraph)
Out{4} >>
executed in 0.00015 [ms]
In{5} << dijkstra(myGraph, 1)
Out{5} >>
```

Itération	Sommet Retiré de L	Couples de sommets				
		1	2	3	4	5
0	-	(0,-)	(∞,-)	(∞,-)	(∞,-)	(∞,-)
1	1	(5,1)	(∞,-)	(∞,-)	(∞,-)	(∞,-)
2	2			(11,2)	(7,2)	(∞,-)
3	4			(11,2)		(11,4)
4	3					(11,4)

```
In{6} <<
```

2 Fonctionnalités

Les fonctionnalités suivantes devront être réalisées avant le 30 mai (semaine 14 du semestre). Les optionnelles seront faites si le temps le permet et les futures sont des améliorations de l'application non prévues dans le cadre du projet.

1. Saisie de graphes orientés et/ou pondérés au moyen d'un langage spécifique.
 - (a) Chargement d'un graphe depuis un fichier externe.
 - (b) ~~(Optionnelle)~~ Génération aléatoire de graphe.
2. Application des algorithmes classiques aux graphes et affichage de leurs résultats.
 - (a) Sauvegarde vers et chargement depuis un fichier texte.
 - (b) Ouverture de plusieurs onglets en parallèle.
 - (c) ~~(Optionnelle)~~ Auto-complétion des fonctions et des variables pendant une saisie.
 - (d) ~~(Optionnelle)~~ Historique des commandes avec les flèches HAUT et BAS.
 - (e) (Future) Exécution/affichage étape par étape des algorithmes.
3. Représentation graphique d'un graphe défini précédemment.
 - (a) Édition au curseur de la représentation graphique, reprise sur tous les dessins.
 - (b) Exportation des dessins au format SVG.
 - (c) (Optionnelle) Exportation dans d'autres formats d'image.
 - (d) (Optionnelle) Exportation de fichier CSV représentant le graphe.
 - (e) (Future) Dessin du graphe directement de manière optimale (avec le moins de chevauchement au niveau des arcs/arêtes).
4. Un menu d'aide contenant les différentes entrées/fonctions possibles avec une documentation associée, ainsi qu'une option de recherche.
 - (a) (Future) Possibilité de lancer un algorithme via une fenêtre de dialogue.

2.1 Famille de graphe

Cette section précise les types de graphe gérés par l'application :

- Orienté / non-orienté
- Biparti / classique
- Sommets pondérés / non-pondérés
- Sommets avec capacités min et max
- Arcs/arêtes pondéré(e)s / non-pondéré(e)s
- Arcs/arêtes avec capacités min et max

2.2 Algorithmes

Cette section précise les familles d'algorithmes prévues dans l'application :

- Parcours du graphe (DFS, BFS)
- Tri topologique
- Détection de cycle
- Composantes fortement connexes (Kosaraju, Tarjan)
- Arbre recouvrant de poids min (Kruskal, Prim)
- Arborescence recouvrante de poids min (Chu-Liu)
- Arborescence recouvrante de section min/max (Prim orienté)

- NB : les noms entre parenthèses sont des exemples d'algorithmes pouvant être implémentés.

- connexe, fortement connexe
- vide

3.1 Diagramme de Gantt



ANNEXE : Journaux de travail

Christopher Browne

Date	Heures	Tâche	Description
21.03.2016	1.0	Modélisation de la classe	Propositions avec le groupe
21.03.2016	0.5	Graphe	Cadre de développement, normalisation du langage, analyse des types et des opérations
21.03.2016	0.5	Organisation groupe	
		Relecture	
23.03.2016	1.0	Étude des widgets Qt	
23.03.2016	2.0	Conception interface aide	
Total	5.0		
04.04.2016	0.5	Documentation	Structure de base du rapport
04.04.2016	1.5	Implémentation interface aide	Création de la GUI pour l'aide
06.04.2016	0.5	Relecture	Grammaire EBNF, Conception de l'interpréteur Suite à un conflit git
08.04.2016	1.0	Relecture tout	
08.04.2016	2.5	Implémentation interface aide	
Total	6.0		
11.04.2016	3.0	Implémentation interface aide	Navigation
11.04.2016	0.5	Relecture	Départ à 0 : utilisation QTextBrowser.
15.04.2015	1.0	Implémentation de la navigation	
Total	4.5		
18.04.2016	1.0	Implémentation interface aide	Ajout barre de recherche et fenêtre principale
18.04.2016	1.0	Conception pages aide	Template HTML
22.04.2016	2.5	Implémentation recherche	Discussion groupe + powerpoint
23.04.2016	2.0	Implémentation recherche	
23.04.2016	2.0	Préparation présentation	
Total	8.5		
25.04.2016	0.5	Discussion groupe	Recherche de la structure adéquate Sous forme de Ternary search Try
25.04.2016	0.5	Présentation intermédiaire	
25.04.2016	2.0	Conception dictionnaires	
29.04.2016	2.0	Implémentation dictionnaire	
Total	5.0		
02.05.2016	1.5	Implémentation dictionnaire	
04.05.2016	1.0	Implémentation dictionnaire	
06.05.2016	1.0	Tests dictionnaire	
06.05.2016	1.0	Corrections dictionnaire	
08.05.2016	1.0	Relecture rapport	
Total	5.5		
09.05.2016	2.5	Implémentation interface aide	Ajout du volet de navigation
13.05.2016	2.5	Rédaction aide utilisateur	

15.05.2016	4.5	Implémentation auto-complétion	Adaptation QCompleter pour le QTextBrowser
Total	9.5		
16.05.2016	4.0	Implémentation auto-complétion	section Interface
18.05.2016	2.0	Rédaction aide utilisateur	
18.05.2016	1.5	Documentation	
20.05.2016	3.5	Rédaction aide utilisateur	
Total	11.0		
23.05.2016	1.5	Aide intégration GUI	Recherche et documentation de bugs Aide utilisateur : problème lié aux signaux/slots
23.05.2016	1.5	Tests GUI/application	
23.05.2016	1.5	Correction auto-complétion	
25.05.2016	5.5	Correction de bugs	
26.05.2016	2.5	Recherche/corrections bugs	
27.05.2016	1.5	Traduction aide utilisateur	
27.05.2016	2.5	Relecture rapport	
29.05.2016	2.5	Traduction aide utilisateur	
29.05.2016	5.0	Mise en commun et documentation	
Total	24.0		
30.05.2016	1.5	Rendu	Impression, reliure...
Total	1.5		
Total	80.5		

Patrick Champion

Date	Heures	Tâche	Description
18.03.2016	3.0	Normalisation du langage	Début de l'analyse
Total	3.0		
21.03.2016	1.5	Normalisation du langage	Suite de l'analyse
23.03.2016	1.5	Normalisation du langage	Suite de l'analyse
24.03.2016	2.0	Normalisation du langage	Grammaire EBNF
Total	5.0		
04.04.2016	2.5	Conception de l'interpréteur	Diagramme de flux des données
05.04.2016	1.5	Conception de l'interpréteur	Flux des données et début de la mémoire virtuelle
06.04.2016	2.0	Conception de l'interpréteur	Conception des types dans la mémoire virtuelle (TDH)
08.04.2016	1.0	Conception de l'interpréteur	Conception des types dans la mémoire virtuelle (Edge et Vertex)
Total	7.0		
14.04.2016	3.0	Conception de l'interpréteur	Tables des variables et des fonctions
15.04.2016	1.5	Conception de l'interpréteur	Transformation et arbre d'appels
16.04.2016	1.5	Conception de l'interpréteur	Modifications dans le rapport et interfage de fonction surchargée
Total	6.0		
18.04.2016	2.0	Conception de l'interpréteur	Fonctions surchargées, vérification des types et début de l'architecture
19.04.2016	1.5	Conception de l'interpréteur	Fin de l'architecture, diagramme de classes

20.04.2016	2.0	Implémentation	Début du parseur
21.04.2016	1.0	Implémentation	Grammaire en cours
22.04.2016	1.0	Implémentation	Grammaire presque finie, à tester
23.04.2016	2.0	Implémentation	Grammaire finie, mais à tester et corriger
24.04.2016	1.5	Implémentation	Grammaire ok mais améliorable, édition du rapport
Total	11.0		
25.04.2016	0.5	Présentation	Présentation intermédiaire de l'état du projet
25.04.2016	1.0	Implémentation	Début de la classe Optional
26.04.2016	3.0	Implémentation	Types ok (Array, Edge, Vertex, Number), VariableTable (manque Graph et TSTMap)
27.04.2016	2.0	Implémentation	FunctionCaller ok, FunctionTable commencée
28.04.2016	1.5	Implémentation	Preprocessor ok, FunctionTable presque finie
29.04.2016	1.0	Implémentation	FunctionTable finie, à tester
Total	9.0		
02.05.2016	1.5	Implémentation	ProcessingUnit commencée
03.05.2016	2.5	Implémentation	ProcessingUnit finie, à tester
04.05.2016	2.0	Implémentation	Début des fonctions 'basics'
Total	6.0		
09.05.2016	2.0	Implémentation	Debugging et fonctions interfaçées
10.05.2016	1.5	Implémentation	Classes Data et Interpreter
11.05.2016	0.5	Discussion	Voir comment on crée des IGraph
12.05.2016	1.5	Implémentation	GraphWrapper et correction de la grammaire (+, -)
13.05.2016	0.5	Implémentation	basics pour les graphes, à impl.
Total	6.0		
19.05.2016	3.0	Implémentation	IGraph* et Vertex insertion
20.05.2016	1.0	Implémentation	toString générique
Total	4.0		
23.05.2016	6.0	Implémentation	GraphWrapper ok, load/save, interfaçage des algos, utilisation dans la GUI
24.05.2016	3.0	Implémentation	Correction de bugs, ajout de la génération aléatoire de graphe, interfaçage
25.05.2016	5.0	Implémentation	Création d'un timer (profiling), performances générales améliorées, interfaçages, ajout de l'export SVG dans la GUI, correction de bugs
26.05.2016	5.0	Implémentation	(Dé-)Sérialisation des données, interfaçage, mise à jour du rapport, performances améliorées (évite certaines copies inutiles), ajout d'une fonction size() pour les graphes et les arrays
27.05.2016	3.0	Implémentation	Correction de la fonction size(), insertions dans GraphWrapper améliorées, correctif sur draw() pour garder la même représentation

28.05.2016	1.0	Documentation	Amélioration du JT, mise à jour du rapport
29.05.2016	6.0	Documentation	Mise à jour du rapport (finale), préparation du déploiement de l'application, derniers tests, relecture, ...
Total	29.0		
30.05.2016	2.0	Rendu	Impression, reliure, ...
Total	2.0		
Total	88.0		

Patrick Djomo

Date	Heures	Tâche	Description
26.03.2016	4.0	Etude des pattern Strategy et Visitor	Découverte des patterns
Total	4.0		
28.03.2016	1.0	Enumération des algos à traiter	Liste des algos à traiter, classés par catégorie
02.04.2016	3.0	Modélisation du diagramme UML	Première ébauche du diagramme en appliquant le pattern Visitor
Total	4.0		
04.04.2016	2.0	Conception de la classe Graph	Différents types de graphes
07.04.2016	2.0	Conception de la classe Graph	Différents types de graphes
Total	4.0		
11.04.2016	4.0	Conception de la classe Graph	Restructuration
Total	4.0		
18.04.2016	4.0	Conception de la classe Graph	Sommets et edges, liste d'adjacence et factories
26.04.2016	4.0	Déclarations des classes et de leurs méthodes	Classes Graph, Vertex, Edge, interface IGraph, FlowGraph, la classe abstraite EdgeCommon et FlowEdge
Total	8.0		
02.05.2016	4.0	Conception	Restructuration de la classe Graph avec Sébastien
Total	4.0		
08.05.2016	4.0	Implémentation classe GraphCommon	Implémentation des méthodes communes à tout type de graphes(celles de la classe CommonGraph)
09.05.2016	4.0	Implémentation classe GraphCommon	Implémentation des méthodes spécifiques aux graphes(celles de la classe Graph)
Total	8.0		
10.05.2016	4.0	Implémentation classe DiGraphCommon	Implémentation des méthodes propres aux graphes orientés et à flow(celles de la classe DiGraphCommon)
11.05.2016	2.0	Implémentation classe DiGraph	Implémentation des méthodes spécifiques au graphes orientés(celles de la classe DiGraph)

12.05.2016	2.0	Implémentation classe FlowGraph	Implémentation des méthodes spécifiques au graphes à flow(celles de la classe DiGraph)
15.05.2016	6.0	Implémentation classe GraphCommon	tests et corrections des bugs
Total	14.0		
16.05.2016	5.0	Implémentation classe	Finalisation avec les classes IEdge, CommonEdge, DiEdgeCommon, Edge et FlowEdge
17.05.2016	2.0	Implémentation classe EdgeCommon	Implémentation des méthodes communes aux edges
18.05.2016	5.0	Implémentation classe Graph	corrections des bugs de conception et des bugs du aux conflits sur gitHub
19.05.2016	5.0	Documentation algorithmes	Recherche et études sur des algorithmes sur la detection des cycles et du tri topologique
20.05.2016	8.0	Implémentations algorithmes	Implémentation des classes DetectedCycle et TopologicalSort ainsi leurs différentes méthodes Ok
21.05.2016	3.0	Implémentations algorithmes	tests des algos detecteCycle et topologiqueSort et création d'une classe pour appeler les algos
Total	28.0		
22.05.2016	3.0	Implémentations algorithmes	Détection et correction d'une erreur logique sur les algo
24.05.2016	1.5	Implémentations algorithmes	Ajout des commentaires sur les classes ainsi que les méthodes au style Qt
25.05.2016	3.0	Implémentations algorithmes	Tests des algo BFS, DFS, BellMan, Dijkstra Ok
26.05.2016	7.0	Implémentations de tout type de graphes	Correctifs apportés aux différentes méthodes des classes afin de réduire certaines complexités
29.05.2016	3.0	Implémentations algorithmes	Implementation de Prim pour les graphes mais test pas Ok
29.05.2016	2.0	Implémentations algorithmes	Implementation de FFEK -FlotMax pour les graphes à flow pas Ok
29.05.2015	2.0	Documentation	cloture avec la section graphe
Total	21.5		
Total	90.5		

Alain Hardy

Date	Heures	Tâche	Description
22.03.2016	1.0	Apprentissage de QtDesigner	Apprentissage et étude des composants mis à disposition par QtDesigner.
23.03.2016	1.0	Apprentissage de QtDesigner	Apprentissage et étude des composants mis à disposition par QtDesigner.

26.03.2016	2.0	Apprentissage de QtDesigner	Apprentissage et étude des composants mis à disposition par QtDesigner.
27.03.2016	2.0	Apprentissage de QtDesigner	Apprentissage et étude des composants mis à disposition par QtDesigner.
Total	6.0		
04.04.2016	1.0	Apprentissage de QtDesigner	Apprentissage et étude des composants mis à disposition par QtDesigner.
05.04.2016	1.0	Apprentissage de QtDesigner	Apprentissage et étude des composants mis à disposition par QtDesigner.
07.04.2016	1.0	Modélisation de l'interface graphique	Réalisation d'une ébauche de l'interface graphique de l'application.
09.04.2016	1.5	Modélisation de l'interface graphique	Réalisation d'une ébauche de l'interface graphique de l'application.
10.04.2016	1.5	Modélisation de l'interface graphique	Réalisation d'une ébauche de l'interface graphique de l'application.
Total	6.0		
11.04.2016	1.5	Création de l'interface graphique	Création de la fenêtre et des composants de base, telle que les onglets de fenêtres et les onglets pour les consoles multiples.
12.04.2016	1.5	Création de l'interface graphique	Début de la création d'une console personnalisée, mise en place de la structure de base.
14.04.2016	1.0	Création de l'interface graphique	Implémentation de la gestion des touches clavier dans la console.
15.04.2016	1.0	Création de l'interface graphique	Implémentation de la gestion des touches clavier dans la console.
16.04.2016	2.0	Création de l'interface graphique	Mise en place de la gestion du curseur afin d'empêcher l'utilisateur d'insérer du texte n'importe où.
Total	7.0		
19.04.2016	2.0	Création de l'interface graphique	Fin de la gestion du curseur et implémentation du déplacement dans la commande via les flèches, et de la suppression de texte.
21.04.2016	1.5	Création de l'interface graphique	Début de l'implémentation de l'historique des commandes. Stockage des commandes lors de leur exécution.
23.04.2016	1.5	Création de l'interface graphique	Navigation dans l'historique des commandes et affichages dans la console de la commande restaurée.
24.04.2016	1.5	Création de l'interface graphique	Implémentation de la création de nouvelle console, ainsi que de la fermeture.
Total	6.5		
25.04.2016	2.0	Création de l'interface graphique	Mise en place d'un menu contextuel personnalisé. Création du widget et de ses composants.

27.04.2016	2.0	Création de l'interface graphique	Création des fonctions pour chacune des actions du menu contextuel.
29.04.2016	2.0	Création de l'interface graphique	Création d'une fenêtr permettant à l'utilisateur de saisir une texte. Sera utilisé pour la création d'une nouvelle console depuis la fenêtr principale.
30.04.2016	1.5	Création de l'interface graphique	Implémentation de la fonctionnalité de création de nouveaux onglets contenant des consoles.
Total	7.5		
02.05.2016	1.5	Création de l'interface graphique	Mise en place de la fermeture des onglets.
06.05.2016	2.0	Modélisation de l'importation/exportation des sessions de travail	Réflexion sur l'implémentation de l'importation/exportation des sessions de travail. Les sessions de travaux étant une sauvegarde toutes les console présentes dans l'applications, il faudrait en premier lieu implémentés la sauvegarde individuelle de console.
06.05.2016	2.0	Modélisation de l'importation/exportation des sessions de travail	Réflexion sur l'implémentation de sauvegarde de console.
Total	5.5		
09.05.2016	2.0	Implémentation de l'importation/exportation des sessions de travail	Implémentation de la sauvegarde de console dans un fichier.
11.05.2016	1.5	Implémentation de l'importation/exportation des sessions de travail	Fin de l'implémentation de la sauvegarde de console dans un fichier.
13.05.2016	1.5	Implémentation de l'importation/exportation des sessions de travail	Implémentation de la restauration de console depuis un fichier.
15.04.2016	1.5	Implémentation de l'importation/exportation des sessions de travail	Finalisation de la restauration de console depuis un fichier.
Total	6.5		
17.05.2016	2.0	Implémentation de l'importation/exportation des sessions de travail	Implémentation de la sauvegarde et restauration de console, à partir des fonctions individuelles des consoles.
19.05.2016	1.0	Tests de l'importation/exportation des sessions de travail	Test sur la sauvegarde et la restauration de console individuelles. Modification du code suite à la découverte de bugs.
20.05.2016	1.0	Tests de l'importation/exportation des sessions de travail	Test sur la sauvegarde et la restauration de session de travail.

21.05.2016	1.0	Tests de l'importation/exportation des sessions de travail	Test sur la sauvegarde et la restauration de session et de console individuelle, afin de voir que l'utilisation des deux fonctionne bien ensemble.
22.05.2016	1.0	Tests de l'importation/exportation des sessions de travail	Test sur la sauvegarde et la restauration de session et de console individuelle, afin de voir que l'utilisation des deux fonctionne bien ensemble.
Total	6.0		
23.05.2016	1.0	Tests de l'importation/exportation des sessions de travail	Test sur la sauvegarde et la restauration de console individuelles. Modification du code suite à la découverte de bugs.
23.05.2016	2.0	Création de l'interface graphique	Création d'un composant permettant l'affichage de graphe.
23.05.2016	2.0	Dessin du graphe avec GUI	Possibilité de dessiner des graphes depuis l'interface graphique.
23.05.2016	1.5	Intégration de l'interpréteur	Intégration de l'interpréteur à la console afin d'exécuter les commandes et d'en recevoir les résultats.
25.05.2016	1.5	Intégration de l'aide utilisateur	Intégration de l'aide utilisateur à l'interface graphique.
25.05.2016	2.0	Exportation SVG avec la GUI	Exportation depuis une commande dans la console de graphe au format SVG.
25.05.2016	2.0	Création de l'interface graphique	Améliorations de l'utilisations de l'interface en intégrant des raccourcis clavier.
26.05.2016	2.0	Tests globaux de l'application	Test d'utilisation de l'applications, notamment la création de graphe, l'affichage, la sauvegarde/restauration de console/session.
26.05.2016	2.0	Documentation	Vérification du code et ajout de commentaires.
29.05.2016	4.0	Documentation	Rédaction de la documentation.
Total	20.0		
Total	71.0		

Sébastien Richoz

Date	Heures	Tâche	Description
25.03.2016	0.5	Etude des pattern Strategy et Visitor	Découverte des patterns
25.03.2016	0.75	Enumération des algos à traiter	Liste des algos à traiter, classés par catégorie
25.03.2016	1.5	Modélisation du diagramme UML	Première ébauche du diagramme en appliquant le pattern Visitor
Total	2.75		
04.04.2016	1.5	Conception de la classe Graph	Différents types de graphes
05.04.2016	1.0	Conception de la classe Graph	Différents types de graphes

Total	2.5		
11.04.2016	1.5	Conception de la classe Graph	Restructuration
Total	1.5		
14.04.2016	3.0	Rapport	Rédaction du chapitre sur l'architecture des graphes
Total	3.0		
18.04.2016	1.5	Conception de la classe Graph	Sommets et edges, liste d'adjacence et factories
24.04.2016	4.0	Déclarations des classes et de leurs méthodes	Classes Graph, Vertex, Edge et interface IGraph
24.04.2016	0.5	Conception du pattern Visitor	Application du pattern Visitor aux graphes
Total	6.0		
25.04.2016	0.5	Présentation	Présentation intermédiaire de l'état du projet
01.05.2016	0.5	Réalisation	BFS non-orienté
01.05.2016	5.0	Réalisation	Corrections de la classe Common-Graph
Total	6.0		
02.05.2016	3.0	Conception	Restructuration de la classe Graph avec Patrick
06.05.2016	0.3	Documentation Algorithmes	Explication sur choix du type de retour des algos
06.05.2016	1.0	Implémentation des algorithmes	DFS non-orienté
Total	4.3		
10.05.2016	1.5	Implémentation classe Graph	Tentative de résolution d'une erreur de compilation
11.05.2016	4.0	Implémentation classe Graph	Finalisation de la classe Graph et tests unitaires de celle-ci
12.05.2016	6.0	Implémentation classe Graph	Fournir une interface IGraph non templée
14.05.2016	2.0	Implémentations algorithmes	Organisation des classes
15.05.2016	2.0	Implémentations algorithmes	tests des algos BFS, DFS, et création d'une classe pour appeler les algos
Total	15.5		
18.05.2016	2.0	Implémentations algorithmes	Kruskal et test
18.05.2016	1.0	Implémentations algorithmes	Prim et test
19.05.2016	3.0	Correctif	Correction de warnings de compilation
19.05.2016	1.0	Implémentations algorithmes	Visiteur pour transformer un graphe en un autre (toutes les combinaisons possibles)
20.05.2016	2.0	Classe graphe	Complétion des classes edges (Edge, DiEdge, FlowEdge)
21.05.2016	2.0	Classe graphe	Complétion des classes graphes (Graph, DiGraph, FlowGraph)

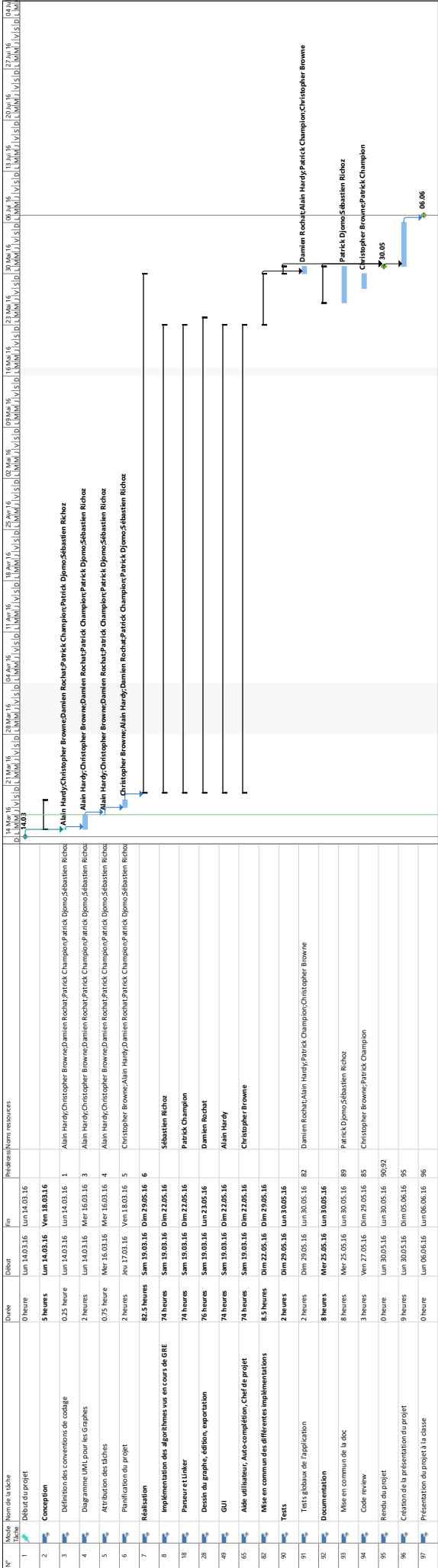
21.05.2016	3.0	Implémentations algo- rithmes	BFS et DFS pour tous types de graphe
22.05.2016	4.0	Implémentations algo- rithmes	Composante connexe, composante for- tement connexe, Dijkstra, Bellman- Ford
Total	18.0		
23.05.2016	8.0	Tests algorithmes	Correctifs amenés à différentes mé- thodes
24.05.2016	2.5	Tests algorithmes	Correction d'un bug sur Prim
25.05.2015	6.0	Implémentations algo- rithmes	Version orientés/non orientées des al- gos
26.05.2015	7.0	Implémentations algo- rithmes	Push-relabel, FFEK (non fonction- nels)
27.05.2015	6.0	Implémentations algo- rithmes	Push-relabel, FFEK (non fonction- nels)
28.05.2015	3.0	Classe graphe	Commentaires et refactoring
28.05.2015	3.0	Documentation	Mise à jour de la section Graphe
29.05.2015	2.0	Documentation	Mise à jour de la section Visiteur
29.05.2015	3.0	Classe Visitor	Commentaires et refactoring du code
29.05.2015	5.0	Implémentation algo- rithmes	Résolution de l'algorithme FFEK
Total	41.5		
Total	101.05		

Damien Rochat

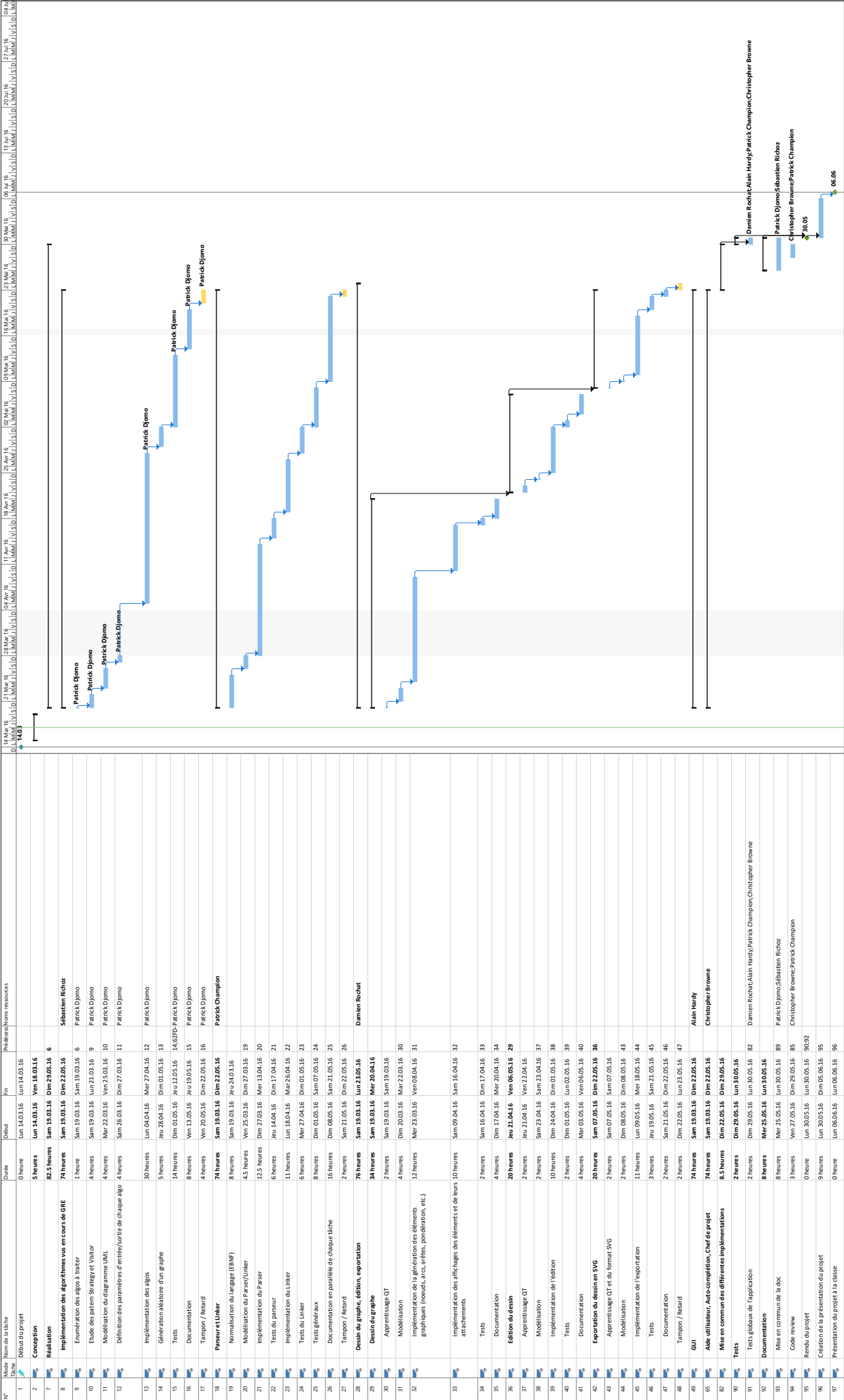
Date	Heures	Tâche	Description
21.03.2016	1.5	Étude de Qt	Apprentissage du système de dessin (View, Scene, Item, etc.)
24.03.2016	4.0	Étude de Qt	Apprentissage du système de dessin (View, Scene, Item, etc.)
Total	5.5		
04.04.2016	2.5	Étude de Qt	Apprentissage du système de dessin (View, Scene, Item, etc.)
08.04.2016	5.0	Étude de Qt	Apprentissage et tests du système de dessin (View, Scene, Item, etc.)
Total	7.5		
11.04.2016	1.5	Dessin du graphe	Implémentation du dessin des som- mets
16.04.2016	6.0	Dessin du graphe	Implémentation du dessin des som- mets
Total	7.5		
18.04.2016	1.5	Dessin du graphe	Implémentation du dessin des arcs et arêtes
21.04.2016	3.0	Dessin du graphe	Implémentation du dessin des arcs et arêtes
	2.5	Dessin du graphe	Petite review du code actuel
Total	7.0		
25.04.2016	0.5	Présentation	Présentation intermédiaire de l'état du projet

	2.0	Edition du graphe	Implémentation du drag&drop des sommets
26.04.2016	4.0	Edition du graphe	Implémentation du drag&drop des sommets
Total	6.5		
02.03.2016	4.0	Edition du graphe	Apprentissage des slots et signaux de QT
	2.0	Edition du graphe	Implémentation du drag&drop des sommets
03.05.2016	3.0	Edition du graphe	Implémentation du drag&drop des sommets
Total	9.0		
10.05.2016	3.0	Exportation SVG	Apprentissage de QT
	2.0	Exportation SVG	Implémentation de l'exportation des graphes en SVG
15.05.2016	4.0	Revue	Refactorisation du code
	2.0	Dessin du graphe	Implémentation des flow graphes
Total	11.0		
17.05.2016	3.5	Interface	Modification du code pour utiliser les classes graphes finales
19.05.2016	2.0	Interface	Liaison avec la GUI
	1.5	Interface	Correction des bugs
21.05.2016	6.0	Dessin du graphe	Correction de bugs, amélioration de l'implémentation des graphes
22.05.2016	6.0	Réutilisation des vues	Implémentation du manageur des vues des graphes permettant la réutilisation de la même vue si la graphe n'a pas changé. Création d'une méthode de hashage des graphes.
Total	19.0		
23.05.2016	3.0	Documentation	Création des diagrammes de classe et rédaction de la documentation
24.05.2016	2.0	Documentation	Création des diagrammes de classe et rédaction de la documentation
	1.5	Correction de bugs	Bug lié à une division par zéro
26.05.2016	3.0	Documentation	Rédaction de la documentation
Total	9.5		
Total final	82.5		

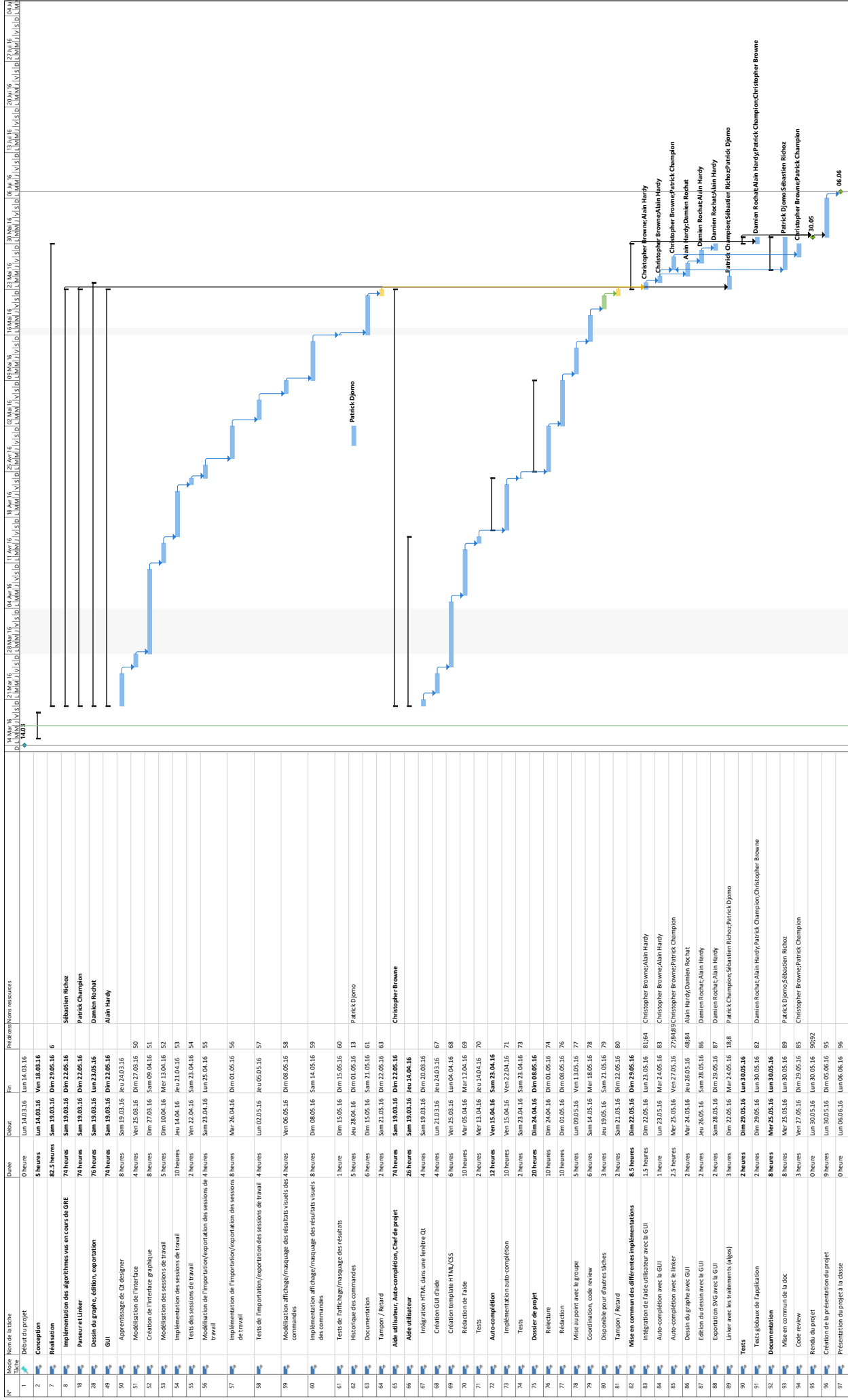
Planification générale



Planification détaillée 1/2




Planification détaillée 2/2



Utilisation des tâches

N°	Mode Tâche	Nom de la tâche	Travail	Durée	Début	Fin	Détails	Tri 2, 2016 Mar	Tri 3, 2016 Jul	Tri 4, 2016 Aoû	Sep	Oct	Nov	Déc	Tri 1, 2017 Jan	Fév	Mar	Tri 2, 2017 Avr	Mai
1		Début du projet		0 heure0 heure	Lun 14.03.16	Lun 14.03.16													
2		Conception		30 heures5 heures	Lun 14.03.16	Ven 18.03.16	Trav.	30h											
3		Définition des conventions de codage		1.5 heures0.25 heure	Lun 14.03.16	Lun 14.03.16	Trav.	1.5h											
4		Diagramme UML pour les Graphes		12 heures2 heures	Lun 14.03.16	Mer 16.03.16	Trav.	12h											
5		Attribution des tâches		4.5 heures0.75 heure	Mer 16.03.16	Mer 16.03.16	Trav.	4.5h											
6		Planification du projet		12 heures2 heures	Jeu 17.03.16	Ven 18.03.16	Trav.	12h											
		<i>Christopher Browne</i>		2 heures	Jeu 17.03.16	Ven 18.03.16	Trav.	2h											
		<i>Patrick Champion</i>		2 heures	Jeu 17.03.16	Ven 18.03.16	Trav.	2h											
		<i>Alain Hardy</i>		2 heures	Jeu 17.03.16	Ven 18.03.16	Trav.	2h											
		<i>Damien Rochat</i>		2 heures	Jeu 17.03.16	Ven 18.03.16	Trav.	2h											
		<i>Patrick Djomo</i>		2 heures	Jeu 17.03.16	Ven 18.03.16	Trav.	2h											
		<i>Sébastien Richaz</i>		2 heures	Jeu 17.03.16	Ven 18.03.16	Trav.	2h											
7		Réalisation		470.5 heures82.5 heures	Sam 19.03.16	Dim 25.05.16	Trav.	78h	188.5h										
8		Implémentation des algorithmes vus en cours de GRE		143 heures74 heures	Sam 19.03.16	Dim 22.05.16	Trav.	26h	53h										
		<i>Sébastien Richaz</i>		74 heures	Sam 19.03.16	Dim 22.05.16	Trav.	13h											
18		Parseur et Linker		74 heures74 heures	Sam 19.03.16	Dim 22.05.16	Trav.	13h	27h										
		<i>Patrick Champion</i>		74 heures	Sam 19.03.16	Dim 22.05.16	Trav.	13h	27h										
28		Dessin du graphe, édition, exportation		76 heures76 heures	Sam 19.03.16	Lun 23.05.16	Trav.	13h	29h										
		<i>Damien Rochat</i>		76 heures	Sam 19.03.16	Lun 23.05.16	Trav.	13h	29h										
49		GUI		79 heures74 heures	Sam 19.03.16	Dim 22.05.16	Trav.	13h	34h										
		<i>Alain Hardy</i>		74 heures	Sam 19.03.16	Dim 22.05.16	Trav.	13h	34h										
65		Aide utilisateur, Auto-complétion, Chef de projet		74 heures74 heures	Sam 19.03.16	Dim 22.05.16	Trav.	13h	27h										
		<i>Christopher Browne</i>		74 heures	Sam 19.03.16	Dim 22.05.16	Trav.	13h	27h										
82		Mise en commun des différentes implémentations		24.5 heures8.5 heures	Dim 22.05.16	Dim 29.05.16	Trav.		24.5h										
90		Tests		8 heures2 heures	Dim 29.05.16	Lun 30.05.16	Trav.		8h										
91		Tests globaux de l'application		8 heures2 heures	Dim 29.05.16	Lun 30.05.16	Trav.		8h										
92		Documentation		22 heures8 heures	Mer 25.05.16	Lun 30.05.16	Trav.		22h										
93		Mise en commun de la doc		16 heures8 heures	Mer 25.05.16	Lun 30.05.16	Trav.		16h										
		<i>Patrick Djomo</i>		8 heures	Mer 25.05.16	Lun 30.05.16	Trav.		8h										
		<i>Sébastien Richaz</i>		8 heures	Mer 25.05.16	Lun 30.05.16	Trav.		8h										
94		Code review		6 heures3 heures	Ven 27.05.16	Dim 29.05.16	Trav.		6h										
95		Rendu du projet		0 heure0 heure	Lun 30.05.16	Lun 30.05.16	Trav.												
96		Création de la présentation du projet		0 heure9 heures	Lun 30.05.16	Dim 05.06.16	Trav.												
97		Présentation du projet à la classe		0 heure0 heure	Lun 06.06.16	Lun 06.06.16	Trav.												

Utilisation des ressources (résumé)

N°		Nom de la ressource	Travail	Détails	Mar	Tri 2, 2016		Mai	Jui	Tri 3, 2016	Aoû	Sep
							Avr			Jul		
		Non affecté	0 heure	Trav.								
1		Christopher Browne	89 heures	Trav.	18h		34h	37h				
2		Patrick Champion	88 heures	Trav.	18h		34h	36h				
3		Alain Hardy	89.5 heures	Trav.	18h		34h	37.5h				
4		Damien Rochat	89 heures	Trav.	18h		34h	37h				
5		Patrick Djomo	87.5 heures	Trav.	18h		34h	35.5h				
6		Sébastien Richoz	87.5 heures	Trav.	18h		34h	35.5h				

Utilisation des ressources (détailé)

N°		Nom de la ressource	Travail	Détails	Mar	Tri 2, 2016 Avr	Mai	Jui	Tri 3, 2016 Jul	Aoû	Sep	Tri 4, 2016 Oct
1		Non affecté		0 heures	Trav.							
		Christopher Browne		89 heures	Trav.	18h	34h	37h				
		Définition des conventions de codage		0.25 heure	Trav.	0.25h						
		Diagramme UML pour les Graphes		2 heures	Trav.	2h						
		Attribution des tâches		0.75 heure	Trav.	0.75h						
		Intégration de l'aide utilisateur avec la GUI		1.5 heures	Trav.			1.5h				
		Auto-complétion avec la GUI		1 heure	Trav.			1h				
		Auto-complétion avec le linker		2.5 heures	Trav.			2.5h				
		Code review		3 heures	Trav.			3h				
		Planification du projet		2 heures	Trav.	2h						
		Aide utilisateur, Auto-complétion, Chef de projet		74 heures	Trav.	13h	34h	27h				
		Tests globaux de l'application		2 heures	Trav.			2h				
2		Patrick Champion		88 heures	Trav.	18h	34h	36h				
		Définition des conventions de codage		0.25 heure	Trav.	0.25h						
		Diagramme UML pour les Graphes		2 heures	Trav.	2h						
		Attribution des tâches		0.75 heure	Trav.	0.75h						
		Parseur et Linker		74 heures	Trav.	13h	34h	27h				
		Auto-complétion avec le linker		1 heure	Trav.			1h				
		Linker avec les traitements (algos)		3 heures	Trav.			3h				
		Tests globaux de l'application		2 heures	Trav.			2h				
		Planification du projet		2 heures	Trav.	2h						
		Code review		3 heures	Trav.			3h				
3		Alain Hardy		89.5 heures	Trav.	18h	34h	37.5h				
		Définition des conventions de codage		0.25 heure	Trav.	0.25h						
		Diagramme UML pour les Graphes		2 heures	Trav.	2h						
		Attribution des tâches		0.75 heure	Trav.	0.75h						
		GUI		74 heures	Trav.	13h	34h	27h				
		Intégration de l'aide utilisateur avec la GUI		1.5 heures	Trav.			1.5h				
		Auto-complétion avec la GUI		1 heure	Trav.			1h				
		Dessin du graphe avec GUI		2 heures	Trav.			2h				
		Edition du dessin avec la GUI		2 heures	Trav.			2h				
		Exportation SVG avec la GUI		2 heures	Trav.			2h				
		Tests globaux de l'application		2 heures	Trav.			2h				
		Planification du projet		2 heures	Trav.	2h						
4		Damien Rochat		89 heures	Trav.	18h	34h	37h				
		Définition des conventions de codage		0.25 heure	Trav.	0.25h						
		Diagramme UML pour les Graphes		2 heures	Trav.	2h						
		Attribution des tâches		0.75 heure	Trav.	0.75h						
		Dessin du graphe, édition, exportation		76 heures	Trav.	13h	34h	29h				
		Edition du dessin avec la GUI		2 heures	Trav.			2h				
		Dessin du graphe avec GUI		2 heures	Trav.			2h				
		Exportation SVG avec la GUI		2 heures	Trav.			2h				
		Tests globaux de l'application		2 heures	Trav.			2h				
		Planification du projet		2 heures	Trav.	2h						
5		Patrick Djomo		87.5 heures	Trav.	18h	34h	35.5h				
		Définition des conventions de codage		0.25 heure	Trav.	0.25h						
		Diagramme UML pour les Graphes		2 heures	Trav.	2h						
		Attribution des tâches		0.75 heure	Trav.	0.75h						
		Linker avec les traitements (algos)		0.5 heure	Trav.			0.5h				
		Enumération des algos à traiter		1 heure	Trav.	1h						
		Etude des pattern Strategy et Visitor		4 heures	Trav.	4h						
		Modélisation du diagramme UML		4 heures	Trav.	4h						
		Définition des paramètres d'entrée/sortie de chaque algo		4 heures	Trav.	4h						
		Implémentation des algos		30 heures	Trav.		30h					
		Tests		14 heures	Trav.			14h				
		Historique des commandes		5 heures	Trav.		4h	1h				
		Tampon / Retard		4 heures	Trav.			4h				
		Documentation		8 heures	Trav.			8h				
		Planification du projet		2 heures	Trav.	2h						
		Mise en commun de la doc		8 heures	Trav.			8h				
6		Sébastien Richoz		87.5 heures	Trav.	18h	34h	35.5h				
		Définition des conventions de codage		0.25 heure	Trav.	0.25h						
		Diagramme UML pour les Graphes		2 heures	Trav.	2h						
		Attribution des tâches		0.75 heure	Trav.	0.75h						
		Implémentation des algorithmes vus en cours de GRE		74 heures	Trav.	13h	34h	27h				
		Linker avec les traitements (algos)		0.5 heure	Trav.			0.5h				
		Planification du projet		2 heures	Trav.	2h						
		Mise en commun de la doc		8 heures	Trav.			8h				